Development and Motion Analysis of Multiple Scaled Autonomous Vehicles

Undergraduate Honors Thesis for the Department of Mechanical and Aerospace Engineering at The Ohio State University

by Daniel Estadt

May 2016

Committee Members

Advisor: Prof. Junmin Wang

Prof. Ryan Harne

Mentor: Zihao Zhu

Abstract

Presently, extensive testing has been conducted with a single autonomous vehicle that interacts solely with dynamics and static objects but not with other autonomous vehicles. An analysis of how autonomous vehicles interact with other autonomous vehicles has been rarely done. This research focuses on the development of multiple, scaled-down, simple autonomous cars using Arduino logic and analyzing how they follow a trajectory. The objective of this research is to avoid unsafe and uncomfortable conditions for the consumer when this technology becomes readily available to the public. This will be done by creating small-scale vehicles capable of implementing object avoidance algorithms similar to a full-scale autonomous vehicle. The first step is to focus on developing a vehicle that can interact with static and dynamic objects which are not autonomous, and successfully avoid them. This is analogous to an autonomous vehicle that can avoid objects similar to the current standard for autonomous vehicles. The final step is to develop a controller to follow a trajectory for future use in a head-on collision test. Advanced scenarios will be researched alongside this project. From this research we have developed three vehicles that meet scalability parameters and have the ability to follow a straight-line trajectory. We have also found that the bicycle model with constant velocity is not sufficient for our needs and must include acceleration. We have successfully built scaled down autonomous vehicles to analyze multiple scenarios, but further design for the system controllers is necessary. This research will allow us to understand if certain safety parameters are necessary for autonomous vehicles while contributing to developing these safety algorithms if necessary.

Acknowledgements

I want to thank Prof. Junmin Wang, Prof. Jinxiang Wang, and Zihao Zhu. I first want to thank Junmin for allowing me to work in his lab and giving me the opportunity to work on my own idea for my research project. Additionally I want to thank him for all his help and support with the project. Next I want to thank Jinxiang for his help in the preliminary stages of the development of the scaled vehicles. Finally I want to thank Zihao for his extensive help on the project and for taking the time to teach the necessary concepts to complete the project.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

An autonomous vehicle is expected to provide a revolutionary way to drive a car in the future, but what exactly is an autonomous vehicle? It is a car that drives entirely on its own and requires the driver to do nothing to operate it. This technology can be extremely beneficial for people by; reducing human-error, giving people the ability to travel when they have lost the ability to drive, and even allowing for efficient long distance traveling. By allowing everyone to operate one it is important that it is as safe as possible.

These vehicles work by using LIDAR (Light and Radar), wheel encoders, and distance sensors as well as other sensing and actuation mechanisms. The LIDAR technology is used to take in data completely around the vehicle and map the surroundings so that the vehicle is able to discern oncoming obstacles and take the appropriate precautions. The wheel encoders are to control the speed of the vehicle so that it can accurately maintain a specific speed or change speed within a reasonable amount of time. Finally the distance sensors are used to sense objects very close to the vehicle that may not be captured by the LIDAR system.

Now with an understanding of what an autonomous vehicle is and how it works the question may be asked, why is this research necessary? While many companies have used resources to develop these vehicles, an analysis of multiple vehicles on the road has been rarely done. Since the intent of these vehicles is to populate the roadways in the future it is important to understand how they react with each other so that the passengers in the vehicle are as safe as possible. In recent news the autonomous Delphi Car and Google Car came close to wrecking into each other but were able to successfully avoid each other [1]. This situation can be analogous to someone walking on the sidewalk and almost running into the other person. While you both may know the best path to take, without physical communication between the both of you, it may be hard to not have some

impulsive movement to avoid each other. This is similar to an autonomous vehicle because the vehicle itself knows the best course of action but may not know what the other vehicle's best course of action is. Analyzing situations such as this could help to understand if vehicle-to-vehicle communication is necessary, if a standard code is needed so that they can properly communicate, or even to see if dangerous situations like this occur.

To complete this research scaled-down vehicles are used. This is because we have the ability to properly scale down the necessary variables that one may see with a car such as the mass, velocity, etc. The reason we do this is because of the cost effectiveness and the time we save. Scaled vehicles are also much easier to test and develop compared to their full-scale counterpart which effectively saves months of time. Additionally, the cost difference is immense and gives us the ability to use cost effective sensors and controllers such as the cost effective Arduino microcontroller. By using cost effective sensors the need to develop our own sensors is unnecessary. We also have the ability to mimic a full-scale autonomous vehicle's sensors with cost effective sensors because our application is on a smaller scale. The forthcoming chapters will detail in full the sensors we used, the development of the scaled cars, and the testing completed to ensure their accuracy.

Chapter 2: Preliminary Work and Construction of the Scaled Vehicle

Introduction

Work was completed to anticipate the current design of the car due to the limitations of the previous models available in the lab. Additionally the sensors and process for coding the vehicle were changed to accommodate future design possibilities. This section will cover the previous vehicle and its limitations, and the components picked for the design.

Previous Vehicle Design

Figure 1 below shows the previous vehicles anticipated for the research of the multiple autonomous vehicles. For ease of purpose I will refer to the red car as the red car and the other car as the long car.

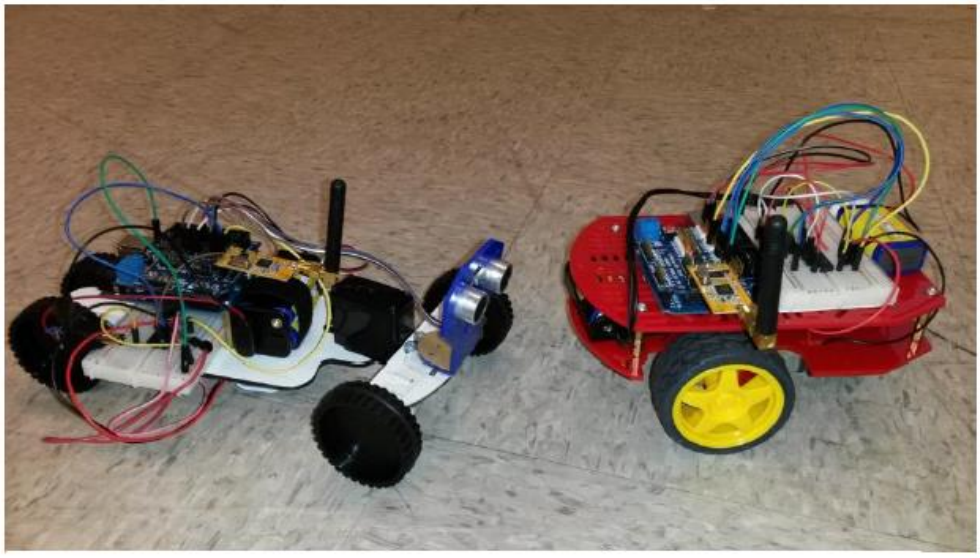


Figure 1: Previous Scaled Vehicle Models

The primary issue with these vehicles was their small design which is something that had to be avoided for research purposes because there was no room on the vehicle to add additional sensors. Due to such limitations we were unable to add multiple distance sensors, accurate wheel encoders, or a gyroscope which were all components necessary for the project. Assuming that adding all of the sensors was possible the weight of all the components would have been well past the scalable mass of the vehicle. Based on our scalability functions, as shown by Equation 2.1 below, and

assuming that the car was modeling a sedan, the necessary mass would have to be 0.107 kg for the red car, and 0.485kg for the long car [2]. The calculation for this value is as follows;

$$m_{scale} = \frac{m_{full-size} * {l_{scale}}^3}{{l_{full-size}}^3} \quad (2.1)$$

Where;

m_{scale} = Mass of scaled vehicle [kg]

$m_{full\text{-}size}$ = Mass of full sized vehicle [kg]

l_{scale} = Length of scaled vehicle [m]

$l_{full\text{-}size}$ = Length of full sized vehicle [m]

The mass of the full sized vehicle is assumed to be an average 4-door sedan which is 1814 kg (4000lbs). The length of a full-size vehicle is assumed to be 4.5m, and the length of the scaled vehicles is 0.175m for the red car and 0.29m for the long car. The actual mass of the red car and long car with just the chassis, distance sensor, two motors, and wheels was 0.34kg and 0.4kg respectively. This means that the red car with only the current sensors fails the scaled vehicle criteria, but the long car does meet the criteria. The issue though still is the space is very limited and also does not include multiple distance sensors, an Arduino microcontroller, an external battery, and the gyroscope. Based on this analysis alone, the previous scaled vehicles had to be scrapped. We did learn that we would have to enlarge the vehicles to accommodate the sensors and be aware of the appropriate weight to model a full scale vehicle. This led to the current design which was significantly larger than the previous models.

Current Vehicle Design

Having found the previous vehicles to be unfit for this research, work began to design a new vehicle for the project. During the summer of 2015 Zihao Zhu took time to develop a new design

for the vehicle which is shown in Figure 2**.** The major advantage of this vehicle is we will be able to place all the necessary sensors on it. Design decisions for this vehicle still had to be made which included choosing the material for the chassis, decision to have shocks, motor size, picking distance sensors, and wheels.

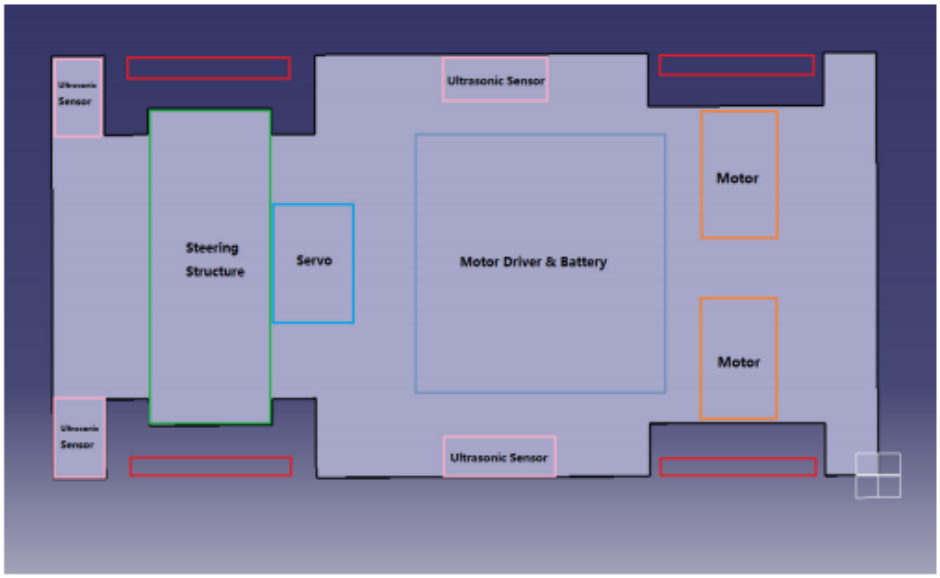


Figure 2: New Scaled Vehicle Design

The first design decision made was picking the material for the chassis. Important factors for choosing the material were that it had to be inexpensive, light-weight, easy to machine, and relatively stiff. Since the chassis is the largest part of the vehicle it was important that it was as light as possible so that the vehicle is able to stay within the mass parameters while having the ability to add as many sensors as necessary. The material also had to be stiff so that we did not introduce additional dynamics into the system that would affect how well we could control the vehicle. An adverse scenario could be the chassis flexing up and down which would disturb the wheels. While metals are typically easy to machine and stiff, it is not very cost effective and would have a significant amount of weight which is why we chose to not consider metal. Another readily available material was an Easy-to-Machine ABS (Acrylonitrile-Butadiene-Styrene) plastic board

from McMaster Carr. The advantages to this material were that it was stiff, inexpensive, easy to machine, and had a low weight. Table 1 below summarizes the differences between the Easy-to-Machine ABS and an Aluminum 6061 sheet. It is entirely possible there is a metal material that would have met our standards, however time was better spent developing the vehicle.

Material	Stiffness [lb/in]	Easy to Machine	Cost	Weight [lbs]
6061 Aluminum	6180	Yes	$82.03	4.51
ABS	124	Yes	$23.95	1.68

Table 1: 6061 Aluminum Sheet Compared to an ABS sheet [3][4]

Each material is available from McMaster Carr and the vehicle has the dimensions of 1/4”x9.45”x19.7”. Additionally the stiffness was calculated by Equation 2.2 below assuming that the vehicle is a fixed beam with a distributed load and rectangular cross section throughout.

$$k = \frac{384EI}{L^3} \tag{2.2}$$

Where;

k = Stiffness [lb/in]

E = Young’s Modulus [psi]

I = Moment of Inertia [in^4]

L = Length [in]

The moment of inertia is calculated as 0.0123 in^4, the length was 19.7”, and the young’s modulus was 1 x 10^7 psi for aluminum and 0.02 x 10^7 psi for ABS. The weight was calculated from density and the volume of the material. From Equation 2.1 we found that the required weight to mimic a full scale car is 5.5 lbs. This can still be met with the ABS material and the deflection from this load would be about 0.04” which is small enough to not cause any issues.

The second design decision to make was whether or not to include shocks on our vehicle. The decision made was that the vehicle would not include shocks for the wheels. This was because we are focused on how the vehicles interact with each other as opposed to the dynamics of the vehicle themselves. This means that on the vehicle we care about the longitudinal and lateral directions and not the vertical axis. Figure 3 shows the coordinate system [5].

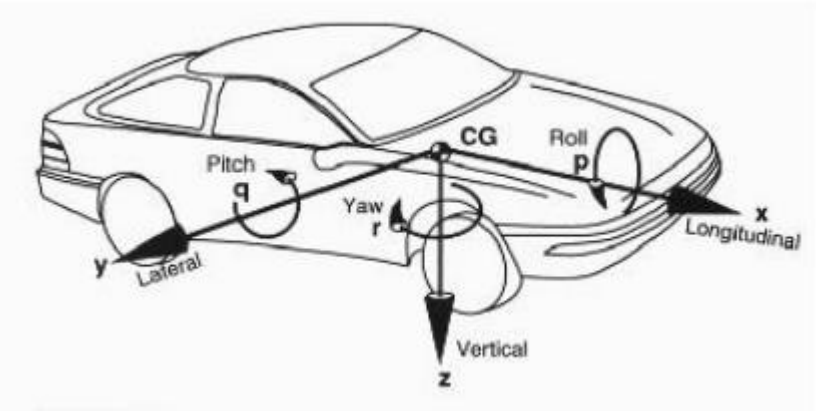


Figure 3: Vehicle Dynamics Axis

By making this decision we do not have to worry about finding compatible parts to include shocks, which makes assembling the vehicle easier.

Following the decision to not include shocks on the car, the motors and wheels were chosen next. For the wheels we used a $1/8^{th}$ scale RC (Remote Control) wheel with a radius of 1.97". We chose this wheel due to its size and because it was a rubber wheel which best imitates a wheel on a full-size car. In order to pick the best motor a torque calculation was required to check that the motor we chose was able to move the vehicle. For this we assumed that the maximum torque would occur when the vehicle would have to overcome static friction. Equation 2.3 used for the analysis is shown below.

$$T_{load} = W\mu r \tag{2.3}$$

Where;

T_{load} = Load Torque [oz-in]

W = Weight [oz]

μ = Static Friction

r = Radius of the Wheel [in]

For the static friction we used a value of 0.5 as this was a value from OSHA for tile floors since the lab we did testing in had tile flooring [6]. The weight of the vehicle was 1.375 lbs (22oz) which came from using the max value calculated from the scalability equation of 5.5lbs divided by 4 due to the vehicle having 4 wheels. The radius is 1.97" coming from our wheel size. The final torque value calculated was 21.67 oz-in which means we needed a motor that could produce above this value. The motor we chose was a 19:1 Metal Gearmotor from Pololu [7]. This motor has a stall torque of 80 oz-in which is well above the maximum value we needed. The motor also came with an encoder which is useful for stabilizing the motor itself using a PID controller to achieve the correct RPM we designate. The max RPM the motor outputs is 500 RPM which is about 103 in/s. Finally a motor driver was picked that would output 12 volts to each motor in case we wanted to operate at its max speed.

The final decision made for the design was picking the correct sensors for the vehicle that would meet our needs. When picking the gyroscope we picked the BNO055 because it interfaced well with the Arduino controller. Picking the distance sensor though was a more complicated process. The initial thought was to use a sensor with one input and output. An image showing one of the sensors considered is shown below in Figure 4 [8].

Figure 4: LV-MaxSonar-EZ0 Sensor

The purpose for having one input and output was to accurately pinpoint an object using the law of cosines. Figure 5 shows this below.

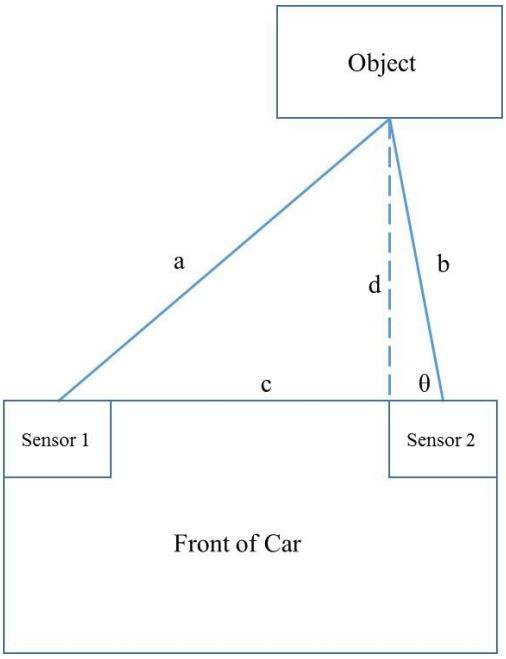


Figure 5: Pinpoint Method

The equation to solve for the variable d is as follows.

$$d = \sin\left(\cos^{-1}\left(\frac{b^2 + c^2 - a^2}{2bc}\right)\right) * b \qquad (2.4)$$

Where;

a = Distance measurement from sensor 1 [cm]

b = Distance measurement from sensor 2 [cm]

c = Length between sensors [cm]

d = Actual distance from object to the front of the vehicle [cm]

While having the ability to pinpoint the location of an object the sensor does not work well between distances of 15 cm to 50.8 cm because of acoustic phase cancellation [8].

The other option was using the ultrasonic sensor HC-SR04. An image showing this is given below in Figure 6 [9].

Figure 6: HC-SR04 Ultrasonic Sensor

The advantage of using this sensor is that it is accurate from 2 cm to 400 cm with a resolution of 3mm at lower distances [9]. This range actually encompassed a large portion of the lab space we designated for the vehicles. This sensor is also very useful because it interfaces easily with the Arduino and works well with multiple distance sensors allowing us to possibly achieve a similar setup to the LIDAR system. The disadvantage with this sensor however is that we cannot pinpoint the location using the two sensors set as shown in the above figure because of a dead zone due to the distance between the input and the output.

For our final decision we ended up picking four of the HR-SC04 ultrasonic sensors because of the high accuracy and for how easy it is to use multiple sensors at once with the Arduino. It is possible

in the future to use both sensors for the application of sensing objects to the side of the car accurately with the pinpoint method. This would be useful because the car would then know if it was okay to merge over into another lane if it were to be running a scenario where it was passing another car. For the time however we will use the HR SC04 for our design.

Now that we have all the components together and the vehicle built we had to check that the final vehicles weight within the scalability boundaries. The mass of the vehicle with all components was 2.27 kg or 5 lbs. Using Equation 2.1 the calculated mass it needed to be was 2.49 kg or 5.5 lbs. While it is still technically under, it is close enough that only a small amount of weight would have to be added to meet the parameter. This also allows us to add additional sensors or components if need be.

With work completed for building the vehicle, the next phase of the project was to test the sensors, testing different codes to run the vehicle, ensure that the vehicle ran properly with the sensors, and describe the kinematics of the vehicle so that we could control it. The current model of the vehicle is included in Figure 7 below.

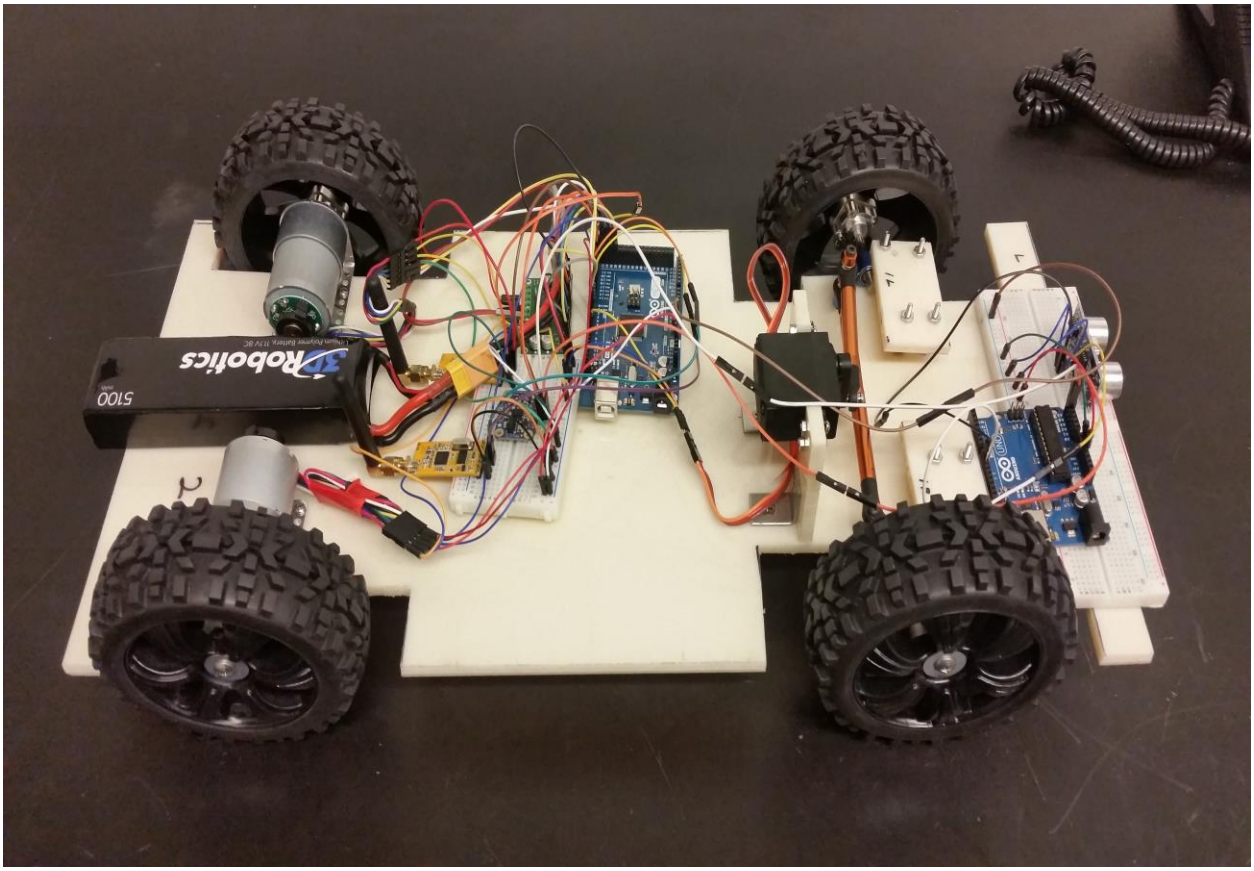


Figure 7: Current Vehicle Model

Chapter 3: Scaled Vehicle Testing

Introduction

With three scaled vehicles built, testing had to be done to check if all the components worked properly and that the vehicles would actually run. To accomplish this goal testing had to be done on the sensors and components. We had to ensure the vehicle ran with all of the sensors, check that code worked properly, and analyze the kinematic model used to describe the vehicle's motion. This section will describe the process taken to test the sensors as well as the results from those tests.

Sensor Testing

The first sensor we tested was the ultrasonic sensor. This was because it was important for our application so that we could actually sense objects nearing the vehicle in order to trigger portions of the code to avoid an object. Currently the sensor is used to detect objects in front of the vehicle due to our focus on preforming object avoidance. Our initial plan for these sensors was to sense and take in data from all sides of the vehicle, however, we ran into a problem with this because of issues with coding the Arduino to work with an array of values.

The first test was to determine sensor accuracy and detect differences between individual sensors. For this test the sensors were placed so that they were steadily facing forward and remained stationary. A cardboard box with a flat face was then placed in front of the sensor at a distance of 50 cm. Since the box was cardboard it should in theory not absorb the 40 kHz signal that the ultrasonic sensor emits. The flat face also makes it easier for the signal to bounce back thus making it ideal to accurately measure the distance. Once several measurement values were taken the box was moved another 50 cm away. Data values were also found to have no fluctuations. This was

done all the way up to 400 cm in increments of 50 cm. The results from this study are shown in Table 2 below.

Actual Distance [cm]	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**392**	**400**
Sensor 1 [cm]	50	100	151	202	254	305	357	400	Error
Sensor 2 [cm]	50	100	151	202	254	305	357	400	Error
Sensor 3 [cm]	50	100	151	202	254	305	357	400	Error
Sensor 4 [cm]	50	100	151	202	254	305	357	400	Error

Table 2: Ultrasonic Distance Accuracy Test

The first thing to notice is that the sensors remain consistent with each other. This will allow us to place the sensors in any position and remain confident that the data will not change due to which sensor we are using. The next thing is that the actual distance away and the measured distance begins to fluctuate at around 150 cm. While a 7cm fluctuation at 350cm is still small, we can use a relationship between the actual distance and the measured distance (correction factor). Equation 3.1 shows this relationship below.

$$y = 0.9753x + 2.2425 \quad (3.1)$$

Where;

y = Actual distance away [cm]

x = Measured distance away [cm]

Having concluded that the distance sensors were accurate and showed no differences between sensors, we were confident that the data taken from the sensors would be sufficient for our project. Our final concern with the device was whether or not additional frequencies would interfere with the device. By looking into the datasheet for the sensor we find that the operating frequency is 40 kHz [9]**.** The only thing to emit this frequency are bats and considering bats do not live in the lab, this was not a concern [10]. It is important to consider that this frequency may also disrupt bats echolocation if an autonomous vehicle were to use ultrasonic sensors sense such as this, however

this is not within the scope of our project. With this final concern accounted for the necessary testing done for the ultrasonic sensors was completed.

For our application we can use the ultrasonic sensors as follows; a way to map the position of the vehicle or to detect objects near the vehicle. Currently the one sensor is used to detect objects in front of it. The purpose of this is to trigger "IF" statements in the Arduino code that tells the vehicle what to do with the oncoming object. This statement currently is to stop the vehicle to ensure that it does not hit the oncoming object. The sensors can also be set up similar to Figure 8 below.

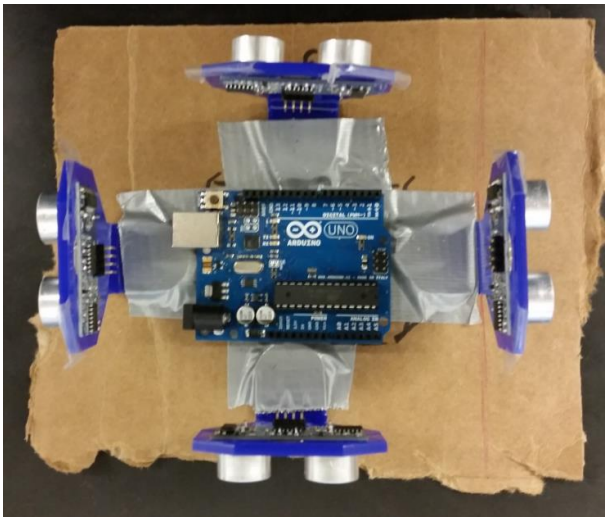


Figure 8: Four Sensor Setup

By setting up the sensors in this way we can map the position of the vehicle relative to its surroundings such as a fenced in area. It would also be able to determine if there are objects close to it and react accordingly. This setup is currently not used since the lab does not have a fenced in area. We do however have the code tested and checked in case it can be used in the future Appendix A.

Component Code Testing:

The focus of this section will be to describe several code algorithms that were developed to test the components and iterations of the code that led to the current design. Codes can be found in Appendix A.

Using the Arduino

For this research we used an Arduino microcontroller to interface with all the sensors and to control the vehicle. The reason for choosing this platform was because of the availability to diagnose issues online, board size range, and previous experience working with an Arduino. To be specific a Mega board and a Uno board were used. In the beginning phases of the research we attempted to use Simulink to program the board's logic, but there were issues with interfacing with the Arduino and the process of programming the logic did not save us time. Therefore it was best to continue the project simply coding the Arduino through the Arduino IDE or through MATLAB.

Servo_Test

This simple code was to test that the servos were working properly. This code was also used to test the min and max angles that we could input for each car. Due to each chassis not following an exact layout as designed (such as bolt holes being slightly off) this test had to be done for each car. The minimum value was the servo value that could be put into the myservo.write command that would hit the side of the chassis. The maximum value was the max value inputted that would hit the other side of the vehicle. Due to the linkage design only integer increments of 5 would show noticeable movement in the angle. The minimum values for Car 1, Car 2, and Car 3 are 70, 70, and 50 respectively. The maximum value for Car 1, Car 2, and Car 3 are 115, 105, and 115 respectively.

Motor_Test

The focus of this test was to make the motors run at the designated speed and check that the wheel encoders were working properly. This code was also used to check that each motor was actually running and that the Arduino and motor driver were wired correctly. A PID controller was used to make the motors run at the desired RPM and the gain values were picked at random until the actual RPM matched the desired RPM.

Research_Code_10-14-15

The purpose of this code was to see if the vehicle would run with all of the sensors and if it would stop at a certain distance away. We found from this design that a mode function was unnecessary for the distance sensors because they were accurate and had no fluctuations in the data. This code also attempted to use multiple distance sensors to achieve the pinpoint method for detecting an object near it. This did not work because the sensors and the vehicle would not work properly and several errors occurred. This was due to several millis functions running in the code which the motors and distance sensor use to operate properly. Due to these errors the vehicle would not run and the distance sensors would not output data correctly. This eventually led us to develop a different approach for implementing the distance sensor which will be discussed in the next section.

Distance_Sensor_Test

This code was used to see if the distance sensors were working and how accurate they were working. This was the code used during the sensor testing mentioned in the above section.

Going_Straight

Due to issues with the distance sensors not operating properly at the time with the vehicle, focus was placed on getting the vehicle to run with the motors and move in a straight line. For this the BNO055 gyroscope was implemented to account for the angular changes. The gyroscope

specifically allowed for easy implementation into the Arduino as there were several libraries already available to allow capture of the values that were needed. The gyroscope was used to stabilize any differences between the two motors. Outside factors would contribute to the vehicle not always moving in a straight line even if the servo was neutral. By using the gyroscope we could measure if any spin was occurring in the vehicle and regulate it to zero using a PID and the servo. Testing was a qualitative analysis, the vehicle would run in a straight line consistently. This also means the vehicle at least has the ability to follow a straight line trajectory in the longitudinal direction. It is important to note this because straight line trajectory planning can be used for a number of applications in autonomous vehicle control such as; following behind a vehicle, staying in a line and not swerving into another lane, and proceeding back in a straight line if it happens to need to turn.

Ensuring the Vehicle Worked with all Sensors

As mentioned previously one of the issues that was occurring was the distance sensor not working properly with the vehicle and causing a number of issues. The distance sensor had to be implemented because the vehicle had to sense it surroundings so that it could react to objects crossing in its path. The primary problem occurring is that an Arduino has one loop function and does not support having two run simultaneously. The distance sensor needs a loop function to trigger and receive distance values and the general code algorithm has to loop at all times. This would lead to the distance sensor trying to obtain measurements and the general code trying to operate as well.

Changing the code to allow everything to operate properly was important because timing and functionality is of the upmost importance. It would not be ideal for an autonomous vehicle to not

be able to make a split second decision because it was stuck in a loop trying to take distance measurements.

To fix this issue, two Arduinos were implemented. The Mega board was still used for all the general code functionality, but a Uno board was used to take in all distance measurement values. This would allow the two loop functions to operate simultaneously and the Mega board to only receive values when it called for them. The initial plan was to sense all four directions as shown in Figure 8, or even the longitudinal and lateral direction, however due to limitations in coding ability the code never functioned well enough to send an array of values. While this may be important in the future, it was deemed not as necessary now since we were only concerned with one direction which is in front of the vehicle. Therefore the Uno board was coded to read one distance measurement in front of the vehicle and send it to the Mega board at a certain distance away when an object was close. This version of the code was a success and consistently sent the value to the Mega board when it was called. Having the ability now to sense an oncoming object now allows for "IF" statements to trigger trajectories to either stop or move the vehicle out of the way. The codes used for this application are included in Appendix A.

State Space Representation

With the vehicle running properly we could now focus on controlling the vehicle itself. This of course is the main focus of autonomous vehicle design and necessary for analyzing multiple autonomous vehicle's interactions. Autonomous vehicle design, requires the vehicle to follow a set trajectory. Even driving on the road a person does trajectory planning by either staying within the lane, moving from behind a slow moving vehicle, planning a proper turn, etc. Every decision made by a person can be related to some trajectory planning and so in order to imitate a person's decision making without the person controlling, the autonomous vehicle must do trajectory

planning. In order to do this the vehicle must be controllable in the longitudinal and lateral plane, as shown by Figure 3**.** Controlling multiple inputs means a state space representation of the vehicles motion must be implemented. Most research applications do this is by assuming a Bicycle Model.

The model we are using to describe the states is shown below in Figure 9 [11].

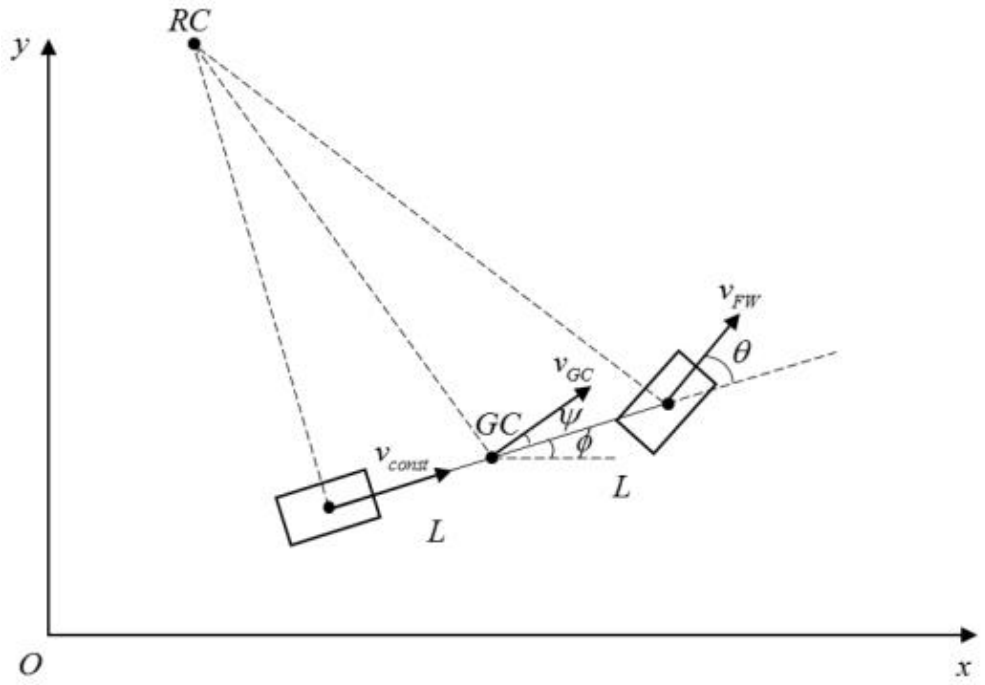


Figure 9: Bicycle Model

Where;

L = Length of the car [m]

$V_{const} = V_{GC} = V_{FW}$ = Velocity [m/s]

Φ = Heading Angle

Ψ = Slip Angle

Θ = Steering Angle

X = Longitudinal Direction

Y = Lateral Direction

The main assumption made here is that the velocity is constant. By making the velocity constant we can reduce the complexity of the design by having no acceleration. The speed is kept low to mitigate any centripetal acceleration when the vehicle turns. The steering angle is controlled by the servo motor, and again has physical limitations. The relatively small steering angle as mentioned in the Servo_Test section means that if the car is turning the centripetal acceleration is mitigated from this as well due to the large turning radius. The heading angle is sensed with the gyroscope which is useful for when we define the equations for the states. The Yaw angle can be written in terms of the heading and steering angle shown by Equations 3.2 – 3.4 below. The states of this system are given as x, y, and φ. The equations representing these states are given by Equations 3.2 - 3.4**,** [11]**.**

$$\dot{x} = V_{GC} * \cos(\Psi + \varphi) = V_{const} * \left(\cos\varphi - \frac{tan\theta}{2} * sin\varphi\right) \tag{3.2}$$

$$\dot{y} = V_{GC} * \sin(\Psi + \varphi) = V_{const} * \left(\sin\varphi + \frac{tan\theta}{2} * cos\varphi\right) \tag{3.3}$$

$$\dot{\varphi} = \omega = V_{const} * \frac{tan\theta}{2L} \tag{3.4}$$

Where the variables are given above.

State Space Analysis

With an understanding of the kinematic model of the vehicle the next step was to control the vehicle. To do this an equation was needed to describe the control input, however the control input in this case was the steering angle. By rewriting the $(tan\theta)/2$ as U we can simplify the above equations. The next step was to develop a relationship with the control input to our state equations. To do this the $\dot{\varphi}$ was measured with the gyroscope at different steering angles.

This test had to be done with each car because of the differences between vehicles. For this test we used the code Angle_Calibration_Test which is shown in Appendix A. The servo value for the car was changed to its minimum value and allowed to move. While the vehicle was moving the rad/s data was collected, and the average value was found. Due to the data having low variance only the average value was taken. Once a run was completed the servo value was increased by 5 and repeated. This was continued until data from the maximum servo value for each car was collected. From these results we obtain the following graphs shown in Figures 10, 11, and 12**.**

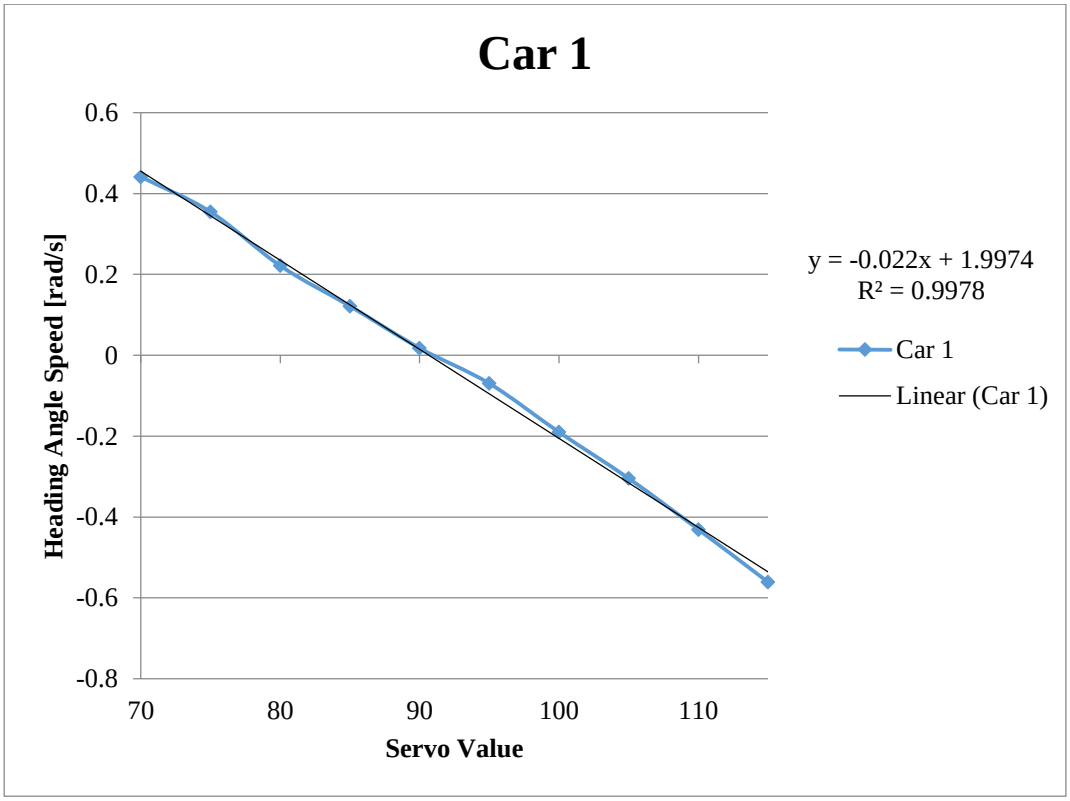


Figure 10: Car 1 Heading Angle Speed vs Servo Value

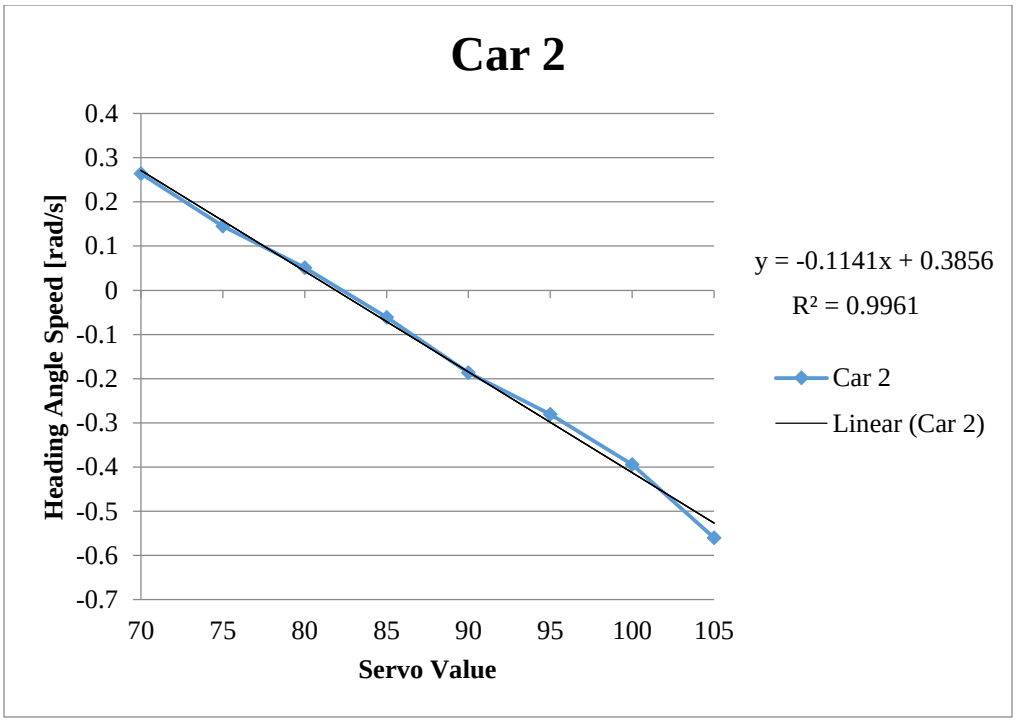


Figure 11: Car 2 Heading Angle Speed vs Servo Value

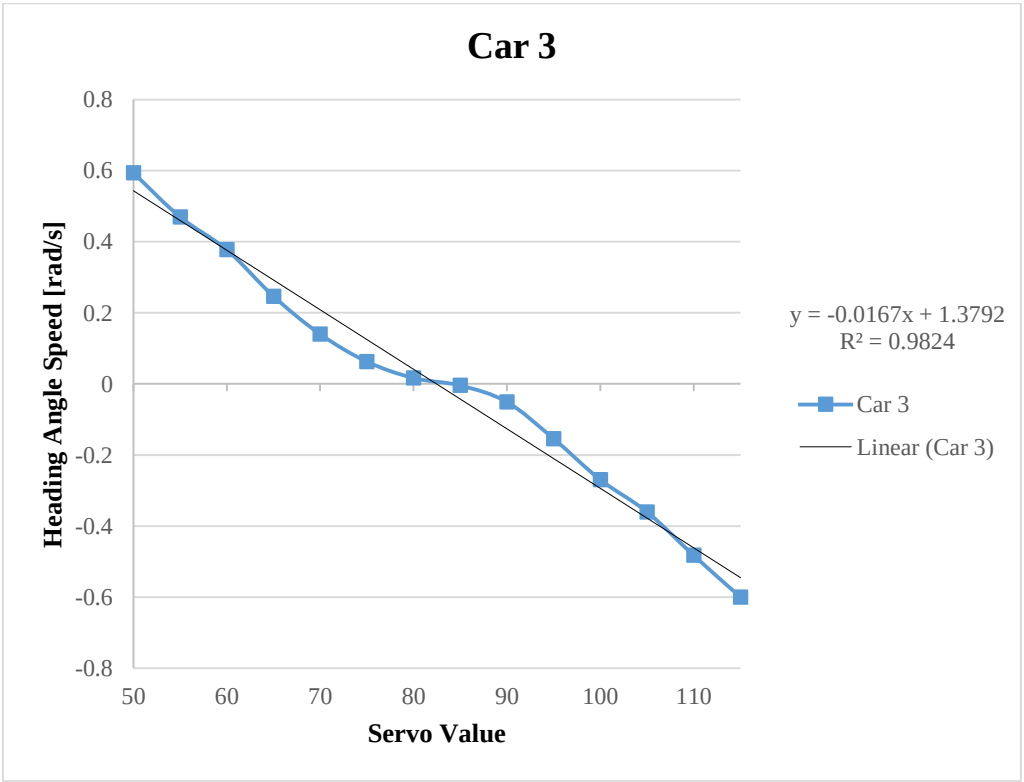


Figure 12: Car 3 Heading Angle Speed vs Servo Value

The results show that we have a linear trend relating the steering angle and speed of the heading angle. It is important to note that both graphs have a high R^2 showing a good fit with the data. From here we can relate the above state equation for $\dot{\varphi}$ to an easy to use equation for our control input. An example of this can be shown by Equation 3.5 for car 1 below.

$$u = \frac{L * (-0.022 * \theta + 1.9974)}{V_{const}} \tag{3.5}$$

Where;

u = Control Input

θ = Steering angle from our servo

V_{const} = Constant Velocity [m/s]

L = Length of the Vehicle [m]

We can now write all of the $tan\theta/2$ in the state space equations with our control input "u". This is valuable because it allows the Arduino to be coded much easier and to build simpler Simulink models to test possible controller designs.

Chapter Conclusion

With the completion of the state space analysis, the development of the vehicle itself is completed. The vehicle now runs properly, is able to detect an object in front of it, and has equations to control its position to allow for trajectory planning. The next phase of the project will be to see if we can design controllers with our vehicle to follow set paths. To do this we will look at the simple case of avoiding a static object. This will let us know if additional work will need to be done to the vehicle to allow it for advanced tracking. The vehicle does well following a straight-line trajectory but in order for advanced roadway scenarios to be analyzed it must be able to turn and avoid an object without having to just brake.

Chapter 4: Trajectory Planning Design

Introduction

Having defined the control input for the vehicle and the state space equations defining the kinematics, the next phase of the project was to see if we could effectively control the vehicle. One of the goals for this project is to avoid an object in front of the vehicle with the trajectory as shown by Figure 13. In this case L_1 = 0.2 m and L_2 = 0.3 m and since the vehicle has to avoid half the distance of L_2 and the full distance of the vehicle itself means the desired final y value should be above 0.35 m. The ability to avoid this object tells us whether or not we can currently control the car in the longitudinal and lateral directions. As mentioned previously the vehicle has the ability to remain in a straight line in the longitudinal direction effectively, but it must be able to operate in both directions. To do this, the controllers must be built using the knowledge we have with the state-space equations mentioned above. This chapter will cover the attempts at controller design and the results found.

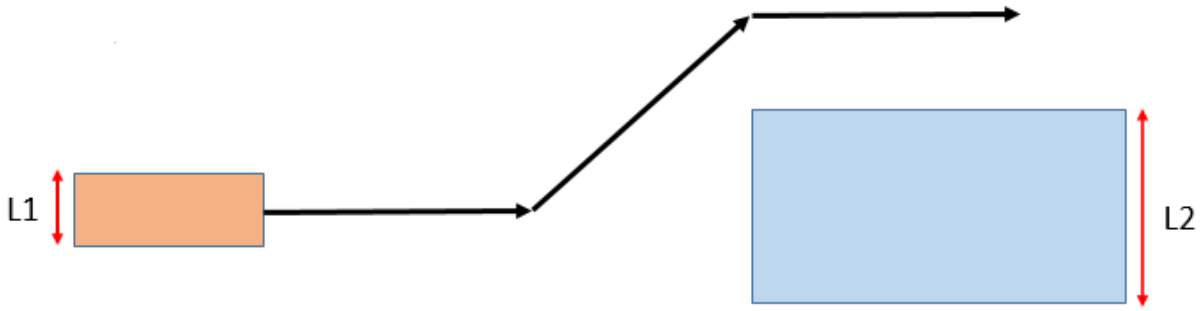


Figure 13: Trajectory Design

Where

L1 = Length of the Front of the car [m]

L2 = Width of the object [m]

Simple Feed-Forward Design

An idea we wanted to analyze was, can we input servo values to achieve the trajectory needed above by already preplanning the path? To save on time we used Simulink to model our system and Feed-Forward design. This is the optimal way to do this because the output of the design can be quickly viewed and we can view all signals with a scope block. Using an Arduino to test designs is not as efficient because time is needed to program the controller, and exporting the data to excel requires time as well. For modeling in Simulink we assume the signal is continuous to simplify the design process. If a controller meets our standards we can model the simulation as discretized points to match the Arduino's 16 MHz clock speed [12]. From here if we still feel the controller is an optimal design we can then code the Arduino to do physical testing. The Simulink diagram for this design is shown below in Figure 14 and 15**.**

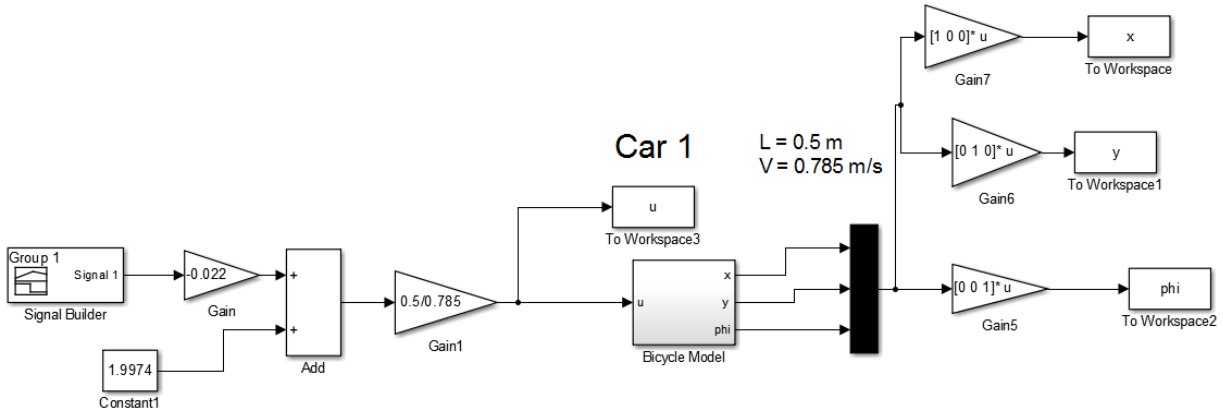


Figure 14: Simulink Model of Bicycle Model

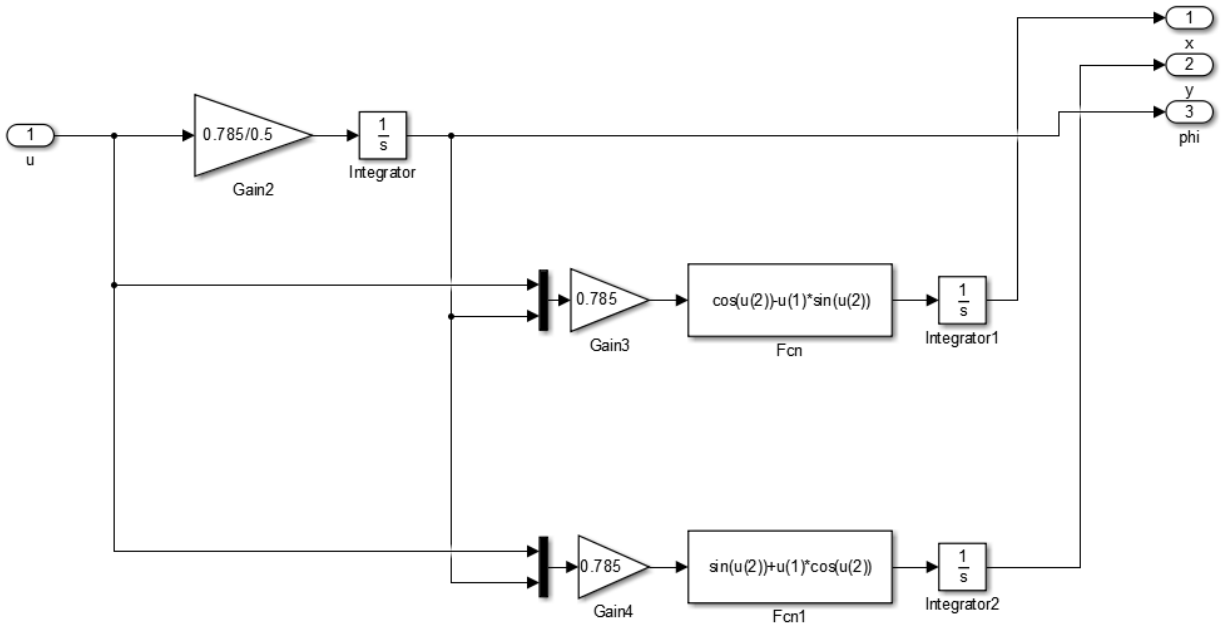


Figure 15: Bicycle Model Subsystem

In this case we are modeling Car 1. The model can be changed to be similar to Car 2 and Car 3 by changing the system input to match the equation we derived earlier from the state space analysis section. To achieve the trajectory as shown the car will have to turn counterclockwise, go straight, turn clockwise, and finally go straight again. We can do this by using a series of "Step" inputs, which is shown by Figure 16 below. There are ramp values to account for the servo taking some time to move to the new angle.

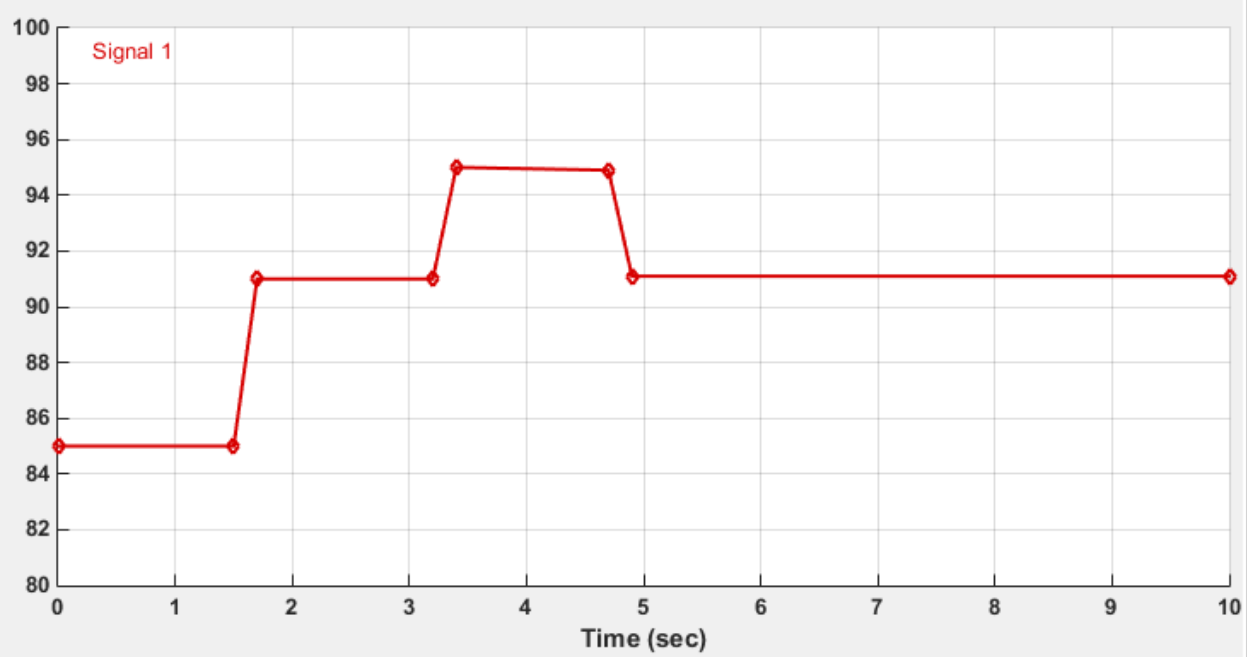


Figure 16: Signal Input

The first value was set at 85 to turn the vehicle counterclockwise. The time spent in the position was varied several times until the output for the desired y was close. The second step value is 91 which was found to be the point at which the vehicle did not turn which can be found by setting the trend lines to 0. This value was held at 1.5 s which can be found using Pythagorean's theorem to find the hypotenuse of a triangle. This hypotenuse was the distance the car needed to travel straight and since we have the speed we can find the time it has to drive straight. In this case the base was 1 m and the height was 0.35 m. The hypotenuse comes out to be 1.06 m and with a velocity of 0.785 m/s the time comes out to be 1.35s which we round to 1.5s to make sure the vehicle actually avoids the object. The third step was 95 to turn the vehicle clockwise. Its time spent here was varied here too. Finally the servo value is set to 90. The output for x, y, phi, and the control input are shown in Figures 17, 18, 19, and 20.

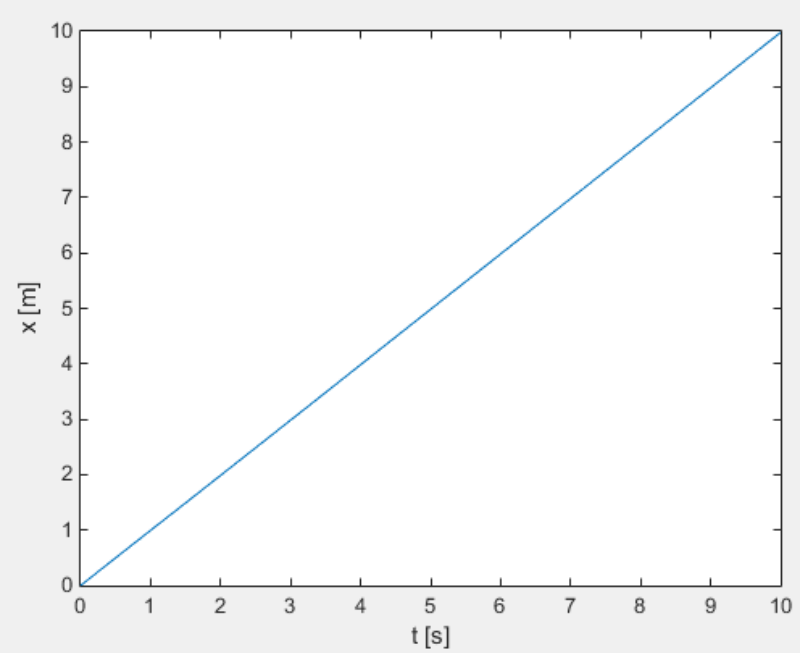


Figure 17: X Output

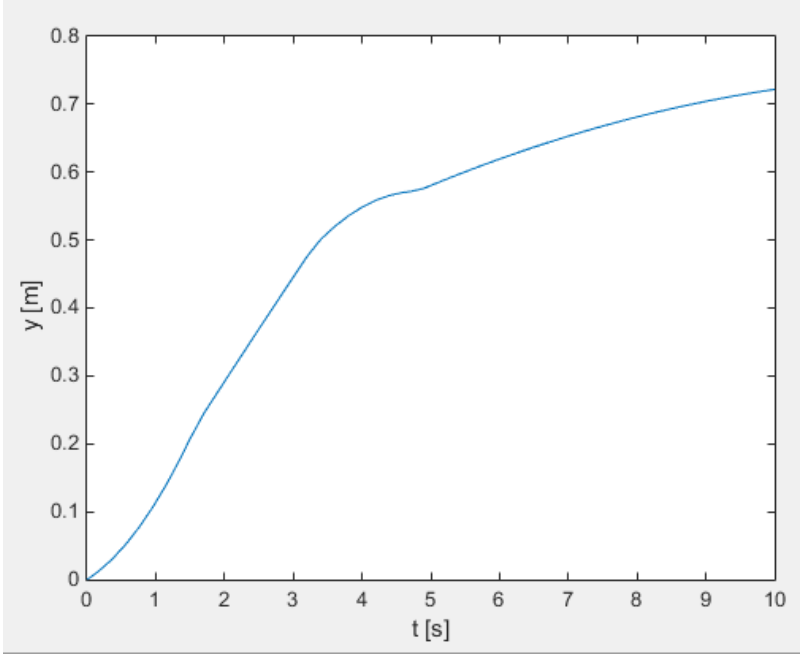


Figure 18: Y Output

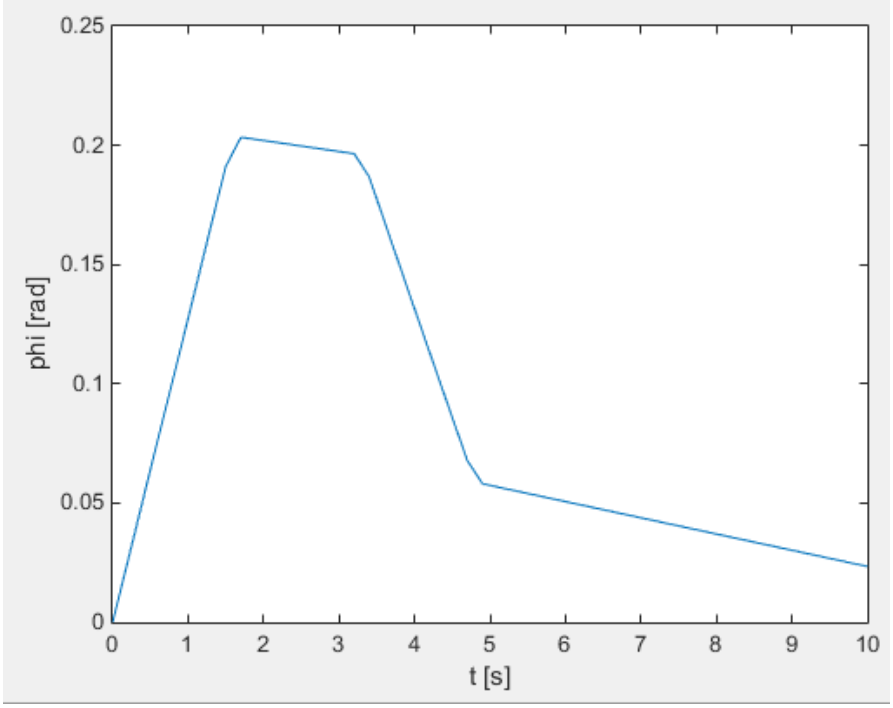


Figure 19: Phi Output

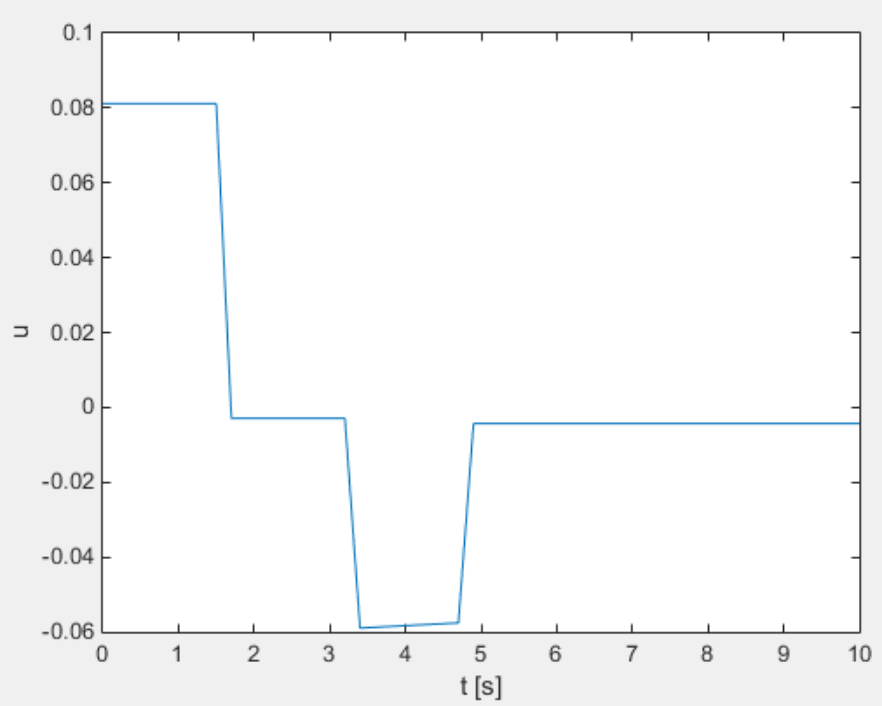


Figure 20: Control Input

Discussion of Results for Simple Feed-Forward

The x (longitudinal) direction acts as expected because it should be constantly moving forward. The y direction (lateral) is able to move to the desired value. The heading angle is close to what we assume because it turns in the positive direction and then goes towards the negative direction. Our control input shows the input we would expect from having a steering angle input in the counterclockwise and clockwise direction. With these results we were curious what would happen if the vehicle continued in what should be a straight line for an additional 90 seconds. What we found is we get undesired trends past 10 seconds. This could be because the servo value for a 0 degree angle is not exactly 91. This works for the short amount of time for the step input used earlier because the error does not propagate as it does when left alone for 90 seconds. Additional servo values were attempted but similar results occurred even when the servo values were 90 or 92. It is important to note as well that the actual vehicle does not produce a noticeable change between servo values that are differences of 1. Figure 21 below shows the accumulation of the results.

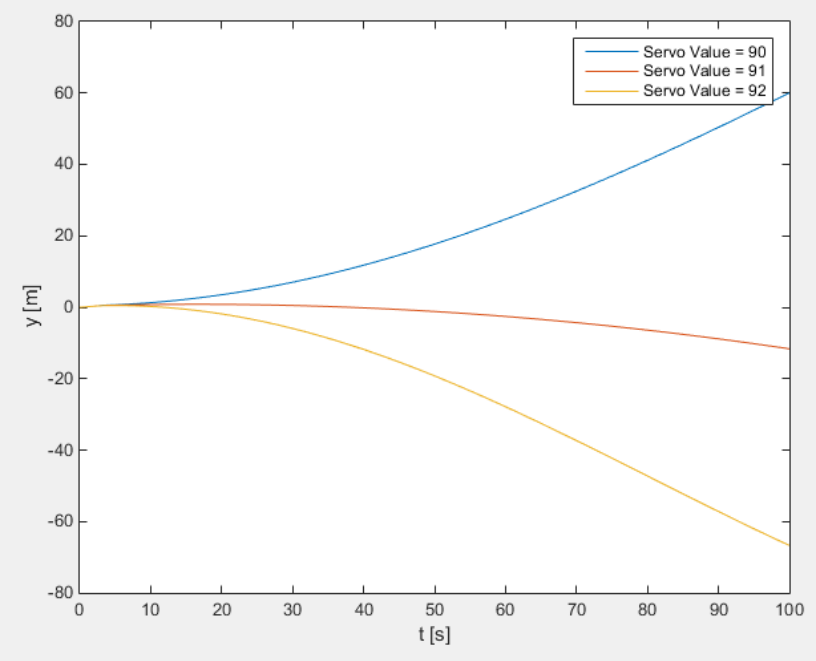


Figure 21: Measured Lateral Distance with Different Servo Values

The key takeaway from this is that feed-forward design not optimal. It is not optimal because significant time was spent attempting to make the car follow the trajectory. Even if the car was able to follow the trajectory, this would have to be changed depending on the size of the object the vehicle was trying to avoid. Finally, having no feedback option causes the errors in the state space to propagate. Therefore it would be best to design a more robust controller to follow the trajectory we need. In order to do this however we would need to linearize our equations because of their non-linear nature. It is important to note that there are still some limitations to linearizing our equations. If we were to drastically move away from our states our controller would become unstable and issues will occur. Additionally a non-linear controller could be designed, but the Arduino cannot handle such a task.

Linearization of State Space Equations

To linearize our state space equations we used a standard linearization method that is based on the Taylor expansion series. We used Equations 4.1 – 4.3 below for our linearization.

$$\dot{x} = V_{GC} * \cos(\Psi + \varphi) = V_{const} * (\cos\varphi - u * sin\varphi) \tag{4.1}$$

$$\dot{y} = V_{GC} * \sin(\Psi + \varphi) = V_{const} * (\sin\varphi + u * cos\varphi) \tag{4.2}$$

$$\dot{\varphi} = \omega = V_{const} * \frac{u}{L} \tag{4.3}$$

The control input equilibrium point was set to 0, and our equilibrium point for φ were 0, $\pi/2$, π, $3\pi/2$. Our general A matrix is given in Equation 4.4 below.

$$A = \begin{bmatrix} \frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial y} & \frac{\partial F_1}{\partial \varphi} \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial y} & \frac{\partial F_2}{\partial \varphi} \\ \frac{\partial F_3}{\partial x} & \frac{\partial F_3}{\partial y} & \frac{\partial F_3}{\partial \varphi} \end{bmatrix} \tag{4.4}$$

Where;

$F_1 = \dot{x}$ Function or Equation 4.1

$F_2 = \dot{y}$ Function or Equation 4.2

$F_3 = \dot{\varphi}$ Function or Equation 4.3

The A matrix filled is as follows in Equation 4.5 below.

$$A = \begin{bmatrix} 0 & 0 & -V * (sin\varphi + u * cos\varphi) \\ 0 & 0 & V * (\cos\varphi - u * sin\varphi) \\ 0 & 0 & 0 \end{bmatrix} \tag{4.5}$$

The general B matrix is as follows in Equation 4.6 below.

$$B = \begin{bmatrix} \frac{\partial F_1}{\partial u} \\ \frac{\partial F_2}{\partial u} \\ \frac{\partial F_3}{\partial u} \end{bmatrix} \tag{4.6}$$

The B matrix filled is as follows in Equation 4.7 below.

$$B = \begin{bmatrix} -V * sin\varphi \\ V * cos\varphi \\ \frac{V}{L} \end{bmatrix} \tag{4.7}$$

From here we find the A and B matrix for each equilibrium point and take the determinant of the controllability and observability matrix to see if we will be able to design controllers from the model. We find that the determinant is 0 for every equilibrium point. This means that based on our state space equations and our current techniques of controller design we are unable to build feedback controllers.

The reason this is occurring could be largely due to assuming that the velocity is constant. Think for example trying to take a turn in a car while going at a constant speed. It is a hard task to do and creates a number of issues which is why people will brake to take this turn. This act of braking however introduces velocity as an input and also introduces acceleration. In most of our cases we try to avoid including acceleration because it complicates the model and that was outside the scope of this project. It is still an important factor to consider in the future but we were focused on simple cases.

Chapter Conclusion

Having determined that our equations were not controllable and that a simple Feed-Forward did not work properly we can conclude that our simplified structure will have to be changed in order to account for trajectory planning. The simple model still has the possibility of a more robust feed-forward controller design but due to time constraints this will not be modeled or tested. The

equations deriving the robust feed-forward design will however be discussed in the next chapter for future work. In addition to the feed-forward design other possibilities will also be discussed.

Chapter 5: Conclusion and Future Work

Having completed some testing on the vehicle and determining that the vehicle was not controllable in its current state there still leaves more work to be done in the future. The vehicle itself drives, and has the possibility of being controllable, however our understanding of the vehicle's states in order to control it need improved. Once the vehicle is controllable, testing roadway scenarios can begin where a significant amount of work will need to be done. This chapter will focus on the next stages for the project and possible ways to improve upon the design. Areas to explore are; testing feed-forward controller designs, adding a sensor for the lateral position, and analyzing the case of a head-on collision.

Feed-Forward with Feedback Controller Design

Currently it is possible to control the vehicle using a feed-forward controller but due to time, testing the relationship to the vehicle states and the control input was not completed. To implement a more robust feed-forward controller we use the following process to develop the necessary gain values to control the vehicle

The equation for the trajectory is given by Equation 5.1**.**

$$y_{desired} = 2 * \left(\frac{\tanh(x_{desired} - 10)}{4} + 1\right) \tag{5.1}$$

Where;

$y_{desired}$ = Desired y values

$x_{desired}$ = Desired x values

The feed forward design comes from the book *Autonomous Ground Vehicles* written by Umit Ozguner, Tankut Acarman, and Keith Redmill [13]. For this case we assume the bicycle model written in terms of the yaw angle and the steering angle as shown below by Equations 5.2 – 5.4.

$$\dot{x} = V_{const} * \cos\left(\Psi + \frac{\theta}{2}\right) \tag{5.2}$$

$$\dot{y} = V_{const} * \sin\left(\Psi + \frac{\theta}{2}\right) \tag{5.3}$$

$$\dot{\Psi} = \omega = V_{const} * \frac{tan\theta}{L_{car}} \tag{5.4}$$

For this method the feed-forward gain comes from a desired R value. Equations 5.5 – 5.10 below are given for this method [13].

$$m_r = \frac{y_B - y_A}{x_B - x_A} \tag{5.5}$$

$$m_t = \frac{y_C - y_B}{x_C - x_B} \tag{5.6}$$

$$X_{FF} = \frac{m_r m_t (y_C - y_A) + m_r (x_B + x_C) - m_t (x_A + x_B)}{2(m_r - m_t)} \tag{5.7}$$

$$Y_{FF} = \frac{y_C + y_A}{2} - \frac{1}{m_t}\left(X_{FF} - \frac{x_B + x_C}{2}\right) \tag{5.8}$$

$$R_{FF} = \sqrt{(x_a - x_{FF})^2 + (y_a - y_{FF})^2} \tag{5.9}$$

$$K_{str-FF} = \tan^{-1}\frac{L_{car}}{R_{FF}} \tag{5.10}$$

Where;

y_b = Current desired y position

y_a = Previous desired y position

x_b = Current desired x position

x_a = Previous desired x position

X_{FF} = Feed forward x position

Y_{FF} = Feed forward y position

R_{FF} = Feed forward radius

$K_{str\text{-}FF}$ = Feed forward gain

L_{car} = Length of car

The gain for the feedback portion of the control loop are given by Equations 5.11 – 5.25 below [13].

$$R_{cp} = \frac{L_{car}}{\tan(\theta_{steer})} \tag{5.11}$$

$$X_{cp} = X_{CG} - \frac{L_{car}}{2} * \sin(\Psi) - R_{cp} * \cos(\Psi) \tag{5.12}$$

$$Y_{cp} = Y_{CG} - \frac{L_{car}}{2} * \cos(\Psi) + R_{cp} * \sin(\Psi) \tag{5.13}$$

$$d_1 = \sqrt{(x_{cp} - x_c)^2 + (y_{cp} - y_c)^2} \tag{5.14}$$

$$\alpha_1 = \frac{({R_{cp}}^2 - {D_{LA}}^2 + {d_1}^2)}{2 * d_1} \tag{5.15}$$

$$h_1 = {R_{cp}}^2 - {\alpha_1}^2 \tag{5.16}$$

$$X_1 = x_c + \alpha_1 * \frac{(x_{cp} - x_c)}{d_1} + h_1 * \frac{(y_{cp} - y_c)}{d_1} \tag{5.17}$$

$$Y_1 = y_c + \alpha_1 * \frac{(y_{cp} - y_c)}{d_1} + h_1 * \frac{(x_{cp} - x_c)}{d_1} \tag{5.18}$$

$$d_2 = \sqrt{(X_{FF} - x_c)^2 + (Y_{FF} - y_c)^2} \tag{5.19}$$

$$\alpha_2 = \frac{({R_{FF}}^2 - {D_{LA}}^2 + {d_2}^2)}{2 * d_2} \tag{5.20}$$

$$h_2 = {R_{FF}}^2 - {\alpha_2}^2 \tag{5.21}$$

$$X_2 = x_c + \alpha_2 * \frac{(x_{FF} - x_c)}{d_2} + h_2 * \frac{(y_{FF} - y_c)}{d_2} \tag{5.22}$$

$$Y_2 = y_c + \alpha_2 * \frac{(y_{FF} - y_c)}{d_2} + h_2 * \frac{(x_{FF} - x_c)}{d_2} \tag{5.23}$$

$$d_{os} = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2} \tag{5.24}$$

$$K_{str-FB} = K_p * d_{os} + K_d * \dot{d_{os}} + K_i * \int d_{os}\, dt \tag{5.25}$$

Where;

R_{cp} = Current position radius

x_{cp} = Current x position

X_{cg} = Center of gravity x position

y_{cp} = Current y position

Y_{cg} = Center of gravity y position

x_c = Next desired x position

d_{os} = Offset distance

By using these gain values some testing can be done through Simulink, followed by programming it onto the vehicle. It would be advised though to calculate the feed forward values through MATLAB first and upload those array values onto the controller to save on the space provided by the Arduino.

Head-on Collision

One of the first cases that can be analyzed is a head-on collision. What is useful about analyzing this case is that you can study directly how two vehicles will interact with each other because the entire decision making process is dependent upon what the other vehicle does. This case also has the ability to increase the difficulty of the scenario by adding walls or another vehicle passing alongside the two vehicles.

The plan for this test is to first see if both vehicles have the ability to avoid the sitting block with the defined trajectory. Once both vehicles are able to do so, simply rerun the code with both the

vehicles in front of each other. The assumption is that the vehicles will avoid each other, however, we are unsure if they will actually do it fast enough. The vehicles can then be coded with more options such as one having the ability to just brake and the other to avoid the vehicle. Further increasing the case the vehicles can start to determine if just braking or moving out of the way is the best option. This final process would be similar to how autonomous vehicles work as they assess their surroundings and pick the best course of action to avoid any issues.

Adding a Sensor for the Lateral Position

One of the improvements that can be added to the vehicle is having a distance sensor read the lateral position of the vehicle. This is important because it reduces the need for an observer controller in the final design. This can be done by adding walls around the vehicles and having it map the position of the vehicle relative to that wall. The need for having the longitudinal direction is not as necessary as the lateral direction because this position is currently being read for triggering the IF statements in the code to avoid the object. This was not implemented at the time due to issues with the Uno sending an array of values to the Mega board. With additional knowledge of Arduino coding this could be done.

Conclusion

Autonomous vehicles will play an important role in the future. Many auto manufactures are already working on having hybrid autonomous vehicles in the next couple of years [14]. Once the first fleet of vehicles hit the road it is important that all considerations are made so that the customer and other vehicles around them are as safe as possible. Someday it is possible that everyone will own one these vehicles, thus marking a shift in how the car will be used.

With this research we have completed important stages to start to analyze how multiple autonomous vehicles work. We now have 3 working vehicles in the lab that have the ability to be programmed for autonomous actions. All 3 vehicles have been tested to that same level as

described in this paper. We have thoroughly detailed the process to build these vehicles so that research can be done easily to analyze the multitude of driving scenarios that can be seen on the road. We also stayed within the necessary parameters to model a full scale vehicle which makes the process of mimicking a full scale vehicle easier. The actual process for building the vehicles can be found in Appendix B.

We have also tested out controller possibilities and found what does not work. Using just a PID with the current states does not effectively control the vehicle. We have also found that the kinematic model with a constant velocity is not sufficient for our case but a feed-forward controller can be tested on the simple model and easily implemented to a more complicated design if need be. We have concluded that more sensors will be needed to sense the additional states of the vehicle such as the lateral position. This will help by not needing to develop an observer controller to determine our unknown states.

Finally we have concluded that there is quite a bit of future work still to be done. There are many different roadway possibilities and ways to improve the control of the vehicle to best mimic the full scale autonomous vehicle. There is an endless amount of research that can be done by studying multiple autonomous vehicles on the road and will be extremely important for improving the understanding of these vehicles. Additionally, by understanding the possible faults of these vehicles we can ensure that they are as safe as possible for the passenger and those around them.

Appendix A: Codes

Angle_Calibration_Test and Going_Straight

Code was used to test the heading angle speed vs servo value and to test if it could move straight. This is because the code is able to stabilize the heading angle which is useful for both applications. Written by Zihao Zhu.

```
// Going Straight.ino

// Information For Gyroscope
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>

Adafruit_BNO055 bno = Adafruit_BNO055(55);

// Information For Servo
#include <Servo.h>
Servo myservo;

// Information For The Encoder
#define left_A 0
#define left_B 1
#define right_A 4
#define right_B 5

static long flag_timer;
static long timer;
static double count_left = 0;
static double count_right = 0;
static double rpm_left = 0;
static double rpm_right = 0;

// Information For The Motor
const int IN1_left = 22;
const int IN2_left = 23;
const int IN1_right = 24;
const int IN2_right = 25;
const int EN = 26;
const int D1_left = 27;
const int D2_left = 4;
const int D1_right = 28;
const int D2_right = 5;

// Information For Heading Angle
static double raw_angle = 0;
```

```
static double angle = 0;
static double angle_control = 0;

// Information For PID Controller
#include <PID_v1.h>

static double angle_desire = 0;
static double servo_voltage_increasement = 0;

double Kp_angle = 5, Ki_angle = 0, Kd_angle = 0;
PID myPID_straight(&angle_control, &servo_voltage_increasement, &angle_desire, Kp_angle,
Ki_angle, Kd_angle, DIRECT);

static double rpm_desire_left = 100;
static double rpm_desire_right = 100;
static double voltage_left = 0;
static double voltage_right = 0;

double Kp_left = 1.3, Ki_left = 3, Kd_left = 0;
PID myPID_left(&rpm_left, &voltage_left, &rpm_desire_left, Kp_left, Ki_left, Kd_left,
DIRECT);

double Kp_right = 1, Ki_right = 3, Kd_right = 0;
PID myPID_right(&rpm_right, &voltage_right, &rpm_desire_right, Kp_right, Ki_right,
Kd_right, DIRECT);

// Information For Longitudinal Distance
static double longitudinal_distance = 0;

void setup() {



/* Preparation Work For Communication */
// Communication Initialization
Serial.begin(9600);
/* Preparation Work For Communication */




/* Preparation Work For Motor */
// Encode Initialization
attachInterrupt(left_A, count_left_A, CHANGE);
attachInterrupt(left_B, count_left_B, CHANGE);
attachInterrupt(right_A, count_right_A, CHANGE);
```

```
attachInterrupt(right_B, count_right_B, CHANGE);

// Timer Initialization
timer = millis();

// Motor Initialization
pinMode(IN1_left, OUTPUT);
pinMode(IN2_left, OUTPUT);
pinMode(IN1_right, OUTPUT);
pinMode(IN2_right, OUTPUT);
pinMode(EN, OUTPUT);
pinMode(D1_left, OUTPUT);
pinMode(D2_left, OUTPUT);
pinMode(D1_right, OUTPUT);
pinMode(D2_right, OUTPUT);

digitalWrite(IN1_left, HIGH);
digitalWrite(IN2_left, LOW);
digitalWrite(IN1_right, HIGH);
digitalWrite(IN2_right, LOW);
digitalWrite(EN, HIGH);
digitalWrite(D1_left, LOW);
digitalWrite(D1_right, LOW);
/* Preparation Work For Motor */



/* Preparation Work For PID Controller */
// PID Controller Initialization
myPID_left.SetMode(AUTOMATIC);
myPID_right.SetMode(AUTOMATIC);
myPID_straight.SetMode(AUTOMATIC);
/* Preparation Work For PID Controller */



/* Preparation Work For Servo */
// Servo Initialization
myservo.attach(9);

// Input Servo Signal For Check, 84 Is Default Value
myservo.write(50);

// Pause For A Second
```

```
delay(1000);

// Initialize The Servo Input As 84-86
myservo.write(86);
/* Preparation Work For Servo */


/* Preparation Work For Gyroscope */
// Initialize The Gyroscope
  if(!bno.begin())
  {
    /* There was a problem detecting the BNO055 ... check your connections */
    Serial.print("Ooops, no BNO055 detected ... Check your wiring or I2C ADDR!");
    while(1);
  }

  bno.setExtCrystalUse(true);
/* Preparation Work For Gyroscope */


/* Waiting For Start */
// Pause For Ten Second
delay(10000);
/* Waiting For Start */


// Flag Timer Initialization
flag_timer = millis();

}

void loop() {

  if(millis() - timer > 30){

  // Reinitializing The Timer
  timer = millis();

  // Print Out The Counter Result
  //Serial.print("count_left is ");
  //Serial.println(count_left);
  //Serial.print("count_right is ");
  //Serial.println(count_right);

  // Calculating Wheel Speed
  calculate_speed();
```

```
  // Compute The Traveled Longitudinal Distance
  longitudinal_distance = longitudinal_distance + ((rpm_left + rpm_right) / 2) * (12 * 3.1415926
/ 2000);

  // Print Out The Distance
  //Serial.print("longitudinal_distance is ");
  //Serial.println(longitudinal_distance);

  // Compute The Voltage Value
  myPID_left.Compute();
  myPID_right.Compute();

  // Input The Voltage Signal (PID)
  analogWrite(D2_left, voltage_left);
  analogWrite(D2_right, voltage_right);

  if(millis() - flag_timer < 3000){

  // Compute The Heading Angle
  compute_heading_angle();

  // We Need To Use If Function To Do The Control
  if(angle >= 0){
    // Denote Angle Control
    angle_control = -angle;

    // Compute Servo Signal For Going Straight
    myPID_straight.Compute();

    // Input The Servo Signal For The Mobile Robot
    myservo.write(86 - servo_voltage_increasement);
  }
  else{
    // Denote Angle Control
    angle_control = angle;

    // Compute Servo Signal For Going Straight
    myPID_straight.Compute();

    // Input The Servo Signal For The Mobile Robot
    myservo.write(86 + servo_voltage_increasement);

  }

  //Serial.print("servo_voltage_increasement is: ");
```

```
  //Serial.println(servo_voltage_increasement);

  }

  else{
     // Set A Concrete Servo Voltage
     myservo.write(100);

     // Get The Information Of Phi Dot
     imu::Vector<3> gyro = bno.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);

     // Print Out Phi Dot
     Serial.print(gyro.z());    //radian per second
     Serial.println("");
  }
  }


}

void count_left_A() {

        count_left++;
}

void count_left_B() {

        count_left++;
}

void count_right_A() {

        count_right++;
}

void count_right_B() {

        count_right++;
}

void calculate_speed() {

  rpm_left = count_left * 5 / 3;
  rpm_right = count_right * 5 / 3;

  count_left = 0;
```

```
  count_right = 0;
}

void compute_heading_angle() {

 // Do A Measurement
  sensors_event_t event;
  bno.getEvent(&event);

  // Drag Out The Raw Angle Information From Gyroscope
  raw_angle = event.orientation.x - 180;  // raw_angle could be positive or negative

  // Manipulate The Raw Angle And Try To Make It Lies In [-180 deg, 180 deg]
  if(raw_angle >= 0){
    angle = raw_angle - 180;
  }
  else{
    angle = raw_angle + 180;
  }

  //Serial.print("angle is ");
  //Serial.println(angle);


}
```

Distance_Sensor_Test

Modified code written by Tim Eckel [15]

```
#include <NewPing.h>

#define TRIGGER_PIN  12
#define ECHO_PIN     11
#define MAX_DISTANCE 200

NewPing sonar(TRIGGER_PIN, ECHO_PIN, MAX_DISTANCE);

void setup() {
  Serial.begin(115200);
}

void loop() {
  delay(50);
  int uS = sonar.ping();
  Serial.print("Ping: ");
  Serial.print(uS / US_ROUNDTRIP_CM);
  Serial.println("cm");
```

```
}
```

Foursensortest

Modified code written by Tim Eckel [15]. This case we used 2 sensors but you can change it to 4 by increasing the Sonar Number and adding them to the sensor object array.

```
#include <NewPing.h>

#define SONAR_NUM     2 // Number or sensors.
#define MAX_DISTANCE 400 // Max distance in cm.
#define PING_INTERVAL 33 // Milliseconds between pings.

unsigned long pingTimer[SONAR_NUM]; // When each pings.
unsigned int cm[SONAR_NUM]; // Store ping distances.
uint8_t currentSensor = 0; // Which sensor is active.

NewPing sonar[SONAR_NUM] = { // Sensor object array.
  NewPing(12,11, MAX_DISTANCE),
  NewPing(10,9, MAX_DISTANCE)
};

void setup() {
  Serial.begin(115200);
  pingTimer[0] = millis() + 75; // First ping start in ms.
  for (uint8_t i = 1; i < SONAR_NUM; i++)
    pingTimer[i] = pingTimer[i - 1] + PING_INTERVAL;
}

void loop() {
  for (uint8_t i = 0; i < SONAR_NUM; i++) {
    if (millis() >= pingTimer[i]) {
      pingTimer[i] += PING_INTERVAL * SONAR_NUM;
      if (i == 0 && currentSensor == SONAR_NUM - 1)
//        oneSensorCycle(); // Do something with results.
      sonar[currentSensor].timer_stop();
      currentSensor = i;
      cm[currentSensor] = 0;
      sonar[currentSensor].ping_timer(echoCheck);
    }
  }
  int x = cm[0];
  int y= cm[1];
  Serial.print(x);
  Serial.print('/');
  Serial.println(y);
}
```

```
void echoCheck() { // If ping echo, set distance to array.
  if (sonar[currentSensor].check_timer())
    cm[currentSensor] = sonar[currentSensor].ping_result / US_ROUNDTRIP_CM;
}

void oneSensorCycle() { // Do something with the results.
  for (uint8_t i = 0; i < SONAR_NUM; i++) {
//    Serial.print(i);
//    Serial.print("=");
    Serial.print(cm[i]);
    Serial.print(",");
  }
  Serial.println();
}
```

Research_Code_10-14-15

Test code that includes a mode function found on the Arduino website. [16]

```
// Motor Control.ino
#include <math.h>
#include <NewPing.h>
// Information For The Encoder
#define left_A 2
#define left_B 3
#define right_A 4
#define right_B 5

static long timer;
static double count_left = 0;
static double count_right = 0;
static double rpm_left = 0;
static double rpm_right = 0;

// Information For The Motor
const int IN1_left = 22;
const int IN2_left = 23;
const int IN1_right = 24;
const int IN2_right = 25;
const int EN = 26;
const int D1_left = 27;
const int D2_left = 2;
const int D1_right = 28;
const int D2_right = 3;

// Information For PID Controller
#include <PID_v1.h>
```

```
static double rpm_desire_left = 30;
static double rpm_desire_right = 30;
static double voltage_left = 0;
static double voltage_right = 0;

double Kp_left = 1, Ki_left = 3, Kd_left = 0;
PID myPID_left(&rpm_left, &voltage_left, &rpm_desire_left, Kp_left, Ki_left, Kd_left,
DIRECT);

double Kp_right = 1, Ki_right = 3, Kd_right = 0;
PID myPID_right(&rpm_right, &voltage_right, &rpm_desire_right, Kp_right, Ki_right,
Kd_right, DIRECT);

// Distance Sensor Information
#define triggerpin1 12
#define echopin1 11
#define triggerpin2 8
#define echopin2 9
NewPing sonar1(12, 11);
NewPing sonar2(8, 9);
double rangevalue1[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0};
double rangevalue2[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0}; //**
long pulse1;
long pulse2;
double modE1;
double modE2;
double l = 3.5; //Distance of two sensors facing the same direction
double x;
double Angle;
double Distance;
int cmconv = 59;
int arraysize = 9; //quantity of values to find the median (sample size). Needs to be an odd
number

void setup() {

  // Communication Initialization
  Serial.begin(9600);

  // Encode Initialization
  attachInterrupt(left_A, count_left_A, CHANGE);
  attachInterrupt(left_B, count_left_B, CHANGE);
  attachInterrupt(right_A, count_right_A, CHANGE);
  attachInterrupt(right_B, count_right_B, CHANGE);
```

```
  // Timer Initialization
  timer = millis();

  // Motor Initialization
  pinMode(IN1_left, OUTPUT);
  pinMode(IN2_left, OUTPUT);
  pinMode(IN1_right, OUTPUT);
  pinMode(IN2_right, OUTPUT);
  pinMode(EN, OUTPUT);
  pinMode(D1_left, OUTPUT);
  pinMode(D2_left, OUTPUT);
  pinMode(D1_right, OUTPUT);
  pinMode(D2_right, OUTPUT);

  digitalWrite(IN1_left, HIGH);
  digitalWrite(IN2_left, LOW);
  digitalWrite(IN1_right, HIGH);
  digitalWrite(IN2_right, LOW);
  digitalWrite(EN, HIGH);
  digitalWrite(D1_left, LOW);
  digitalWrite(D1_right, LOW);

  // PID Controller Initialization
  myPID_left.SetMode(AUTOMATIC);
  myPID_right.SetMode(AUTOMATIC);

}

void loop() {
  distance();
  Serial.println(modE2);
  if (modE2 < 4) {
    rpm_desire_left = 0;
    rpm_desire_right = 0;
    motor_control();
    distance();
  }
  if (modE2 > 4) {
    rpm_desire_left = 30;
    rpm_desire_right = 30;
    motor_control();
    distance();
  }

}
```

```
void count_left_A() {

  count_left++;
}

void count_left_B() {

  count_left++;
}

void count_right_A() {

  count_right++;
}

void count_right_B() {

  count_right++;
}

void calculate_speed() {

  rpm_left = count_left * 5 / 3;
  rpm_right = count_right * 5 / 3;

  count_left = 0;
  count_right = 0;
}

void motor_control() {
  if (millis() - timer > 30) {

    // Reinitializing The Timer
    timer = millis();

    // Print Out The Counter Result
    //Serial.print("count_left is ");
    //Serial.println(count_left);
    //Serial.print("count_right is ");
    //Serial.println(count_right);

    // Calculating Wheel Speed
    calculate_speed();

    // Print Out The Speed
    //Serial.print("rpm_left is ");
```

```
    //Serial.println(rpm_left);
//    Serial.print("rpm_right is ");
//    Serial.println(rpm_right);

    // Input The Voltage Signal (Constant)
    //analogWrite(D2_left, 150);
    //analogWrite(D2_right, 150);

    // Compute The Voltage Value
    myPID_left.Compute();
    myPID_right.Compute();

    // Input The Voltage Signal (PID)
    analogWrite(D2_left, voltage_left);
    analogWrite(D2_right, voltage_right);

  }
}

void distance() {

  //  for(int i = 0; i < arraysize; i++)
  //  {
  //    rangevalue1[i] = sonar1.ping_in();
  //    delay(10);
  //  }
  for (int i = 0; i < arraysize; i++)
  {
    rangevalue2[i] = sonar2.ping_in();
    delay(50);
  }

  modE1 = mode(rangevalue1, arraysize); //**
  modE2 = mode(rangevalue2, arraysize); //**
  //  x=((modE1*modE1)+(l*l)-(modE2*modE2))/(2*modE1*l);  //Using the law of cosines to
find the exact distance
  //  Angle=acos(x);  //angle from law of cosines used to find the exact distance
  //  Distance=modE1*sin(Angle);  //Exact distance
  Distance = modE2;
}

//Mode function, returning the mode or median.
double mode(double *x, int n) {

  int i = 0;
  int count = 0;
```

```
  int maxCount = 0;
  double mode = 0;
  int bimodal;
  int prevCount = 0;
  while (i < (n - 1)) {
    prevCount = count;
    count = 0;
    while (x[i] == x[i + 1]) {
      count++;
      i++;
    }
    if (count > prevCount & count > maxCount) {
      mode = x[i];
      maxCount = count;
      bimodal = 0;
    }
    if (count == 0) {
      i++;
    }
    if (count == maxCount) { //If the dataset has 2 or more modes.
      bimodal = 1;
    }
    if (mode == 0 || bimodal == 1) { //Return the median if there is no mode.
      mode = x[(n / 2)];
    }
    return mode;
  }
}
```

Master_Code_Test_Uno_Board

```
//IMPORTANT: DO NOT ADD ANY DELAYS
#include <Wire.h>

#include <NewPing.h>

NewPing sonar(10, 9, 400);

void setup() {
  Serial.begin(9600);

  Wire.begin(0x08);                // join i2c bus with address #8
  Wire.onRequest(requestEvent); // register event
}

void loop() {
  int y = sonar.ping_cm();
```

```
  Serial.println(y);
delay(100);
}



// function that executes whenever data is requested by master
// this function is registered as an event, see setup()
void requestEvent() {
  byte y = sonar.ping_cm();
  Wire.write(y);
}
```

Slave_Receiver_Test_Mega_Board

```
#include <Wire.h>


void setup() {
  Wire.begin();        // join i2c bus (address optional for master)
  Serial.begin(9600);  // start serial for output
}

void loop() {
  Wire.requestFrom(0x08,1);    // request 6 bytes from slave device #8

  while (Wire.available()) { // slave may send less than requested
    int x = Wire.read(); // receive a byte as character
    Serial.println(x);         // print the character
  }

}
```

Motor_Test

Testing that the motors run properly written by Zihao Zhu.

```
// Motor Control.ino

// Information For The Encoder
#define left_A 2
#define left_B 3
#define right_A 4
#define right_B 5

static long timer;
static double count_left = 0;
static double count_right = 0;
static double rpm_left = 0;
```

```
static double rpm_right = 0;

// Information For The Motor
const int IN1_left = 22;
const int IN2_left = 23;
const int IN1_right = 24;
const int IN2_right = 25;
const int EN = 26;
const int D1_left = 27;
const int D2_left = 4;
const int D1_right = 28;
const int D2_right = 5;

// Information For PID Controller
#include <PID_v1.h>

static double rpm_desire_left = 150;
static double rpm_desire_right = 150;
static double voltage_left = 0;
static double voltage_right = 0;

double Kp_left = 1, Ki_left = 3, Kd_left = 0;
PID myPID_left(&rpm_left, &voltage_left, &rpm_desire_left, Kp_left, Ki_left, Kd_left,
DIRECT);

double Kp_right = 1, Ki_right = 3, Kd_right = 0;
PID myPID_right(&rpm_right, &voltage_right, &rpm_desire_right, Kp_right, Ki_right,
Kd_right, DIRECT);


void setup() {

// Communication Initialization
Serial.begin(9600);

// Encode Initialization
attachInterrupt(left_A, count_left_A, CHANGE);
attachInterrupt(left_B, count_left_B, CHANGE);
attachInterrupt(right_A, count_right_A, CHANGE);
attachInterrupt(right_B, count_right_B, CHANGE);

// Timer Initialization
timer = millis();

// Motor Initialization
pinMode(IN1_left, OUTPUT);
```

```
pinMode(IN2_left, OUTPUT);
pinMode(IN1_right, OUTPUT);
pinMode(IN2_right, OUTPUT);
pinMode(EN, OUTPUT);
pinMode(D1_left, OUTPUT);
pinMode(D2_left, OUTPUT);
pinMode(D1_right, OUTPUT);
pinMode(D2_right, OUTPUT);

digitalWrite(IN1_left, HIGH);
digitalWrite(IN2_left, LOW);
digitalWrite(IN1_right, HIGH);
digitalWrite(IN2_right, LOW);
digitalWrite(EN, HIGH);
digitalWrite(D1_left, LOW);
digitalWrite(D1_right, LOW);

// PID Controller Initialization
myPID_left.SetMode(AUTOMATIC);
myPID_right.SetMode(AUTOMATIC);

}

void loop() {

  if(millis() - timer > 30){

  // Reinitializing The Timer
  timer = millis();

  // Print Out The Counter Result
  //Serial.print("count_left is ");
  //Serial.println(count_left);
  //Serial.print("count_right is ");
  //Serial.println(count_right);

  // Calculating Wheel Speed
  calculate_speed();

  // Print Out The Speed
  Serial.print("rpm_left is ");
  Serial.println(rpm_left);
  Serial.print("rpm_right is ");
  Serial.println(rpm_right);

  // Input The Voltage Signal (Constant)
```

```
  //analogWrite(D2_left, 150);
  //analogWrite(D2_right, 150);

  // Compute The Voltage Value
  myPID_left.Compute();
  myPID_right.Compute();

  // Input The Voltage Signal (PID)
  analogWrite(D2_left, voltage_left);
  analogWrite(D2_right, voltage_right);

}

}

void count_left_A() {

  count_left++;
}

void count_left_B() {

  count_left++;
}

void count_right_A() {

  count_right++;
}

void count_right_B() {

  count_right++;
}

void calculate_speed() {

  rpm_left = count_left * 5 / 3;
  rpm_right = count_right * 5 / 3;

  count_left = 0;
  count_right = 0;
}
```

Servo_Test

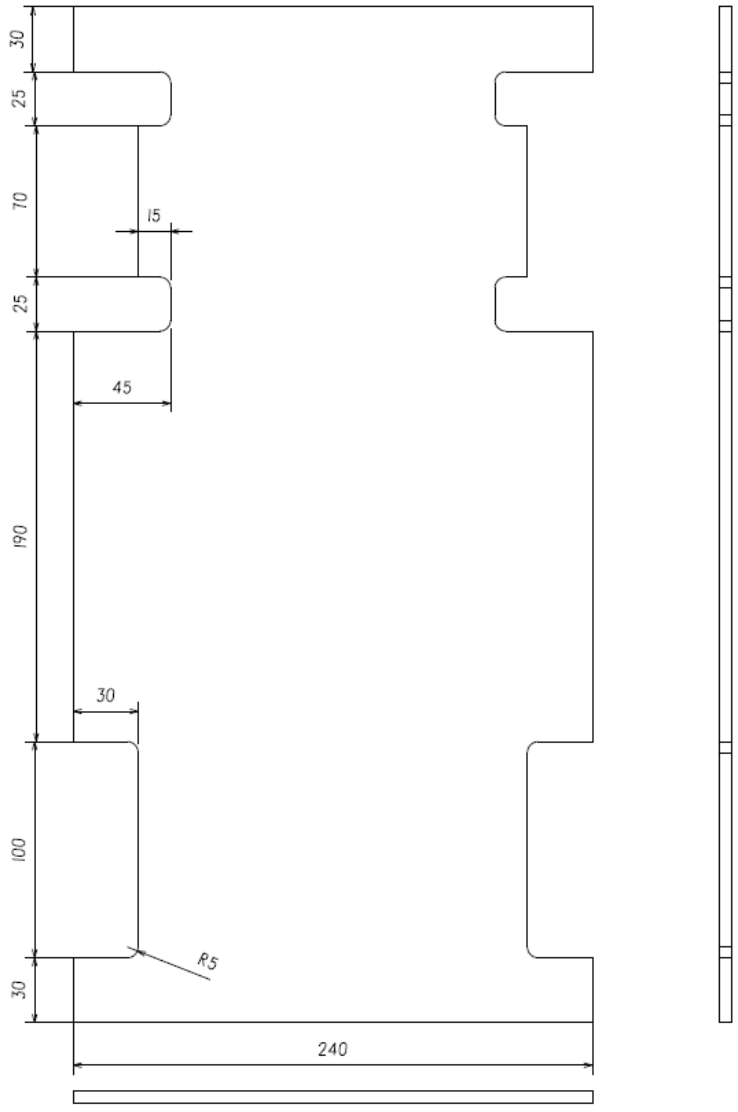
```
// Servo_Control.ino
```

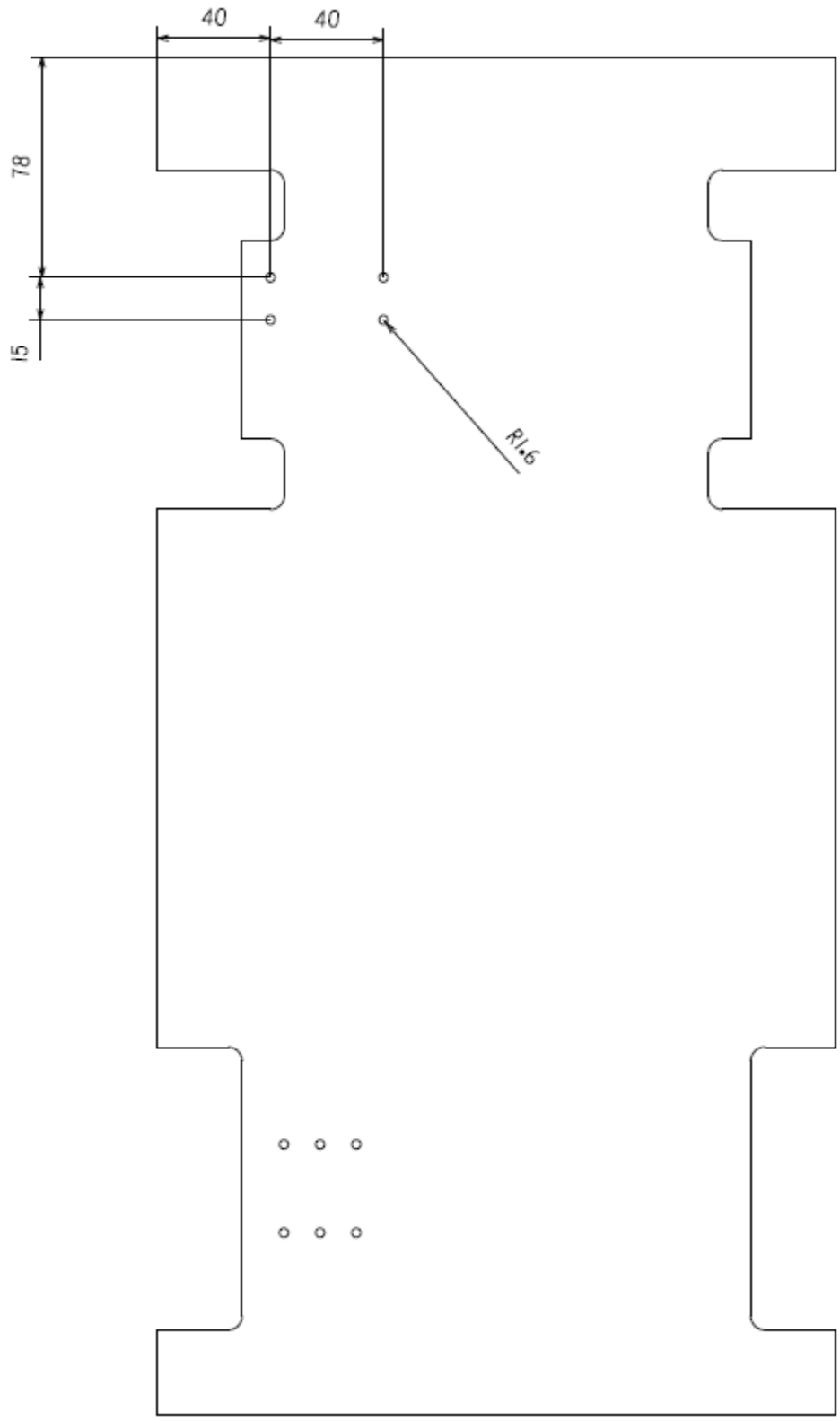
```
// Information For Servo
#include <Servo.h>

Servo myservo;


void setup() {

// Servo Initialization
myservo.attach(9);

}

void loop() {

// Input Servo Signal, 84 Is Default Value
myservo.write(115);

}
```

Appendix B: Detailed Guide for Scaled Vehicle Development

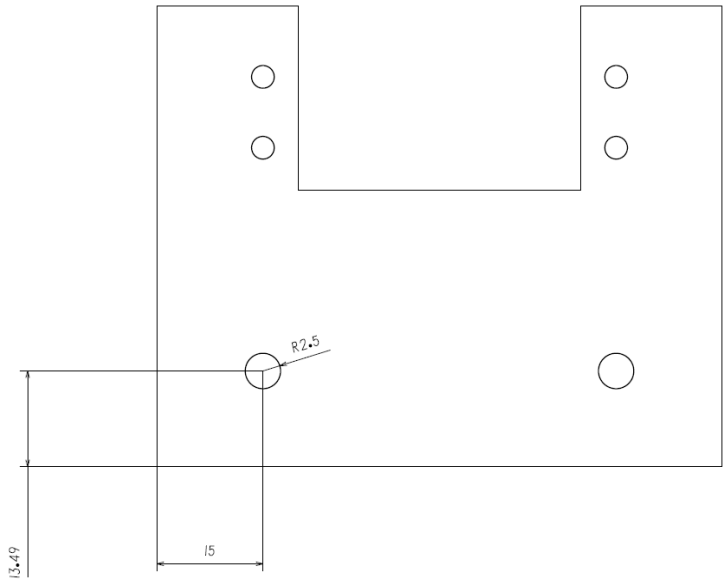
This portion will cover how to build the vehicles described in this thesis. The chassis and the supports were designed by Zihao Zhu.

1. Machine the following Chassis. Modifications may need to be made to fit the chosen wheels. All dimensions are in mm and all holes have a diameter of 3.2 mm. Holes are also mirrored on the right side in the same positions. Additional holes for the servo mount is dependent on the position of the wheels and the linkages.

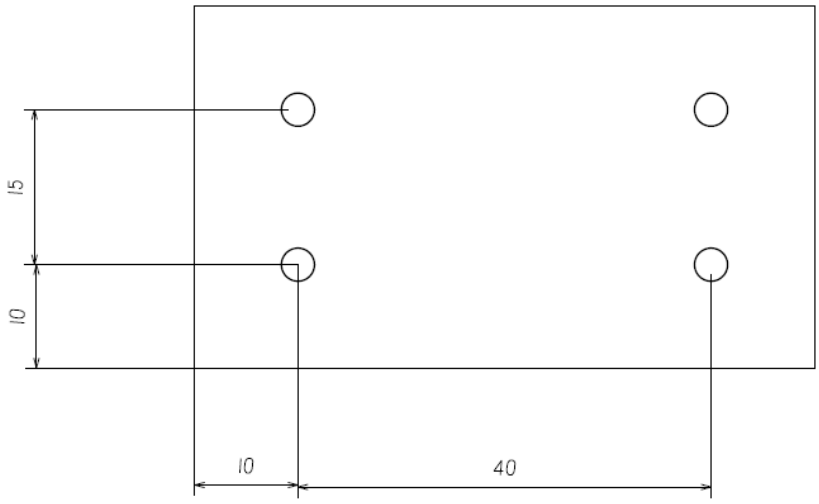




2. Machine the following servo mount. Top holes depend on the size of your servo, for our vehicle it was 3.2 mm in diameter.



3. Machine the steering mount. We used leftover material from the chassis cut out which was 6.35 mm thick.



4. The motor mount and motors we used were Stamped Aluminum L-Bracket Pair for 37D mm Metal Gearmotors and 19:1 Metal Gearmotor with 64 CPR Encoder from Pololu. [17][7].
5. The servo was a Power HD High-Torque Servo from Pololu [18]
6. Next we used 17 mm Hex Wheel Adaptors, 0.770” Clamping Hubs, and 0.770” Set Screw Hubs from Servocity to mount the 1/8th Scale Off Road Car Buggy RC Tires from Amazon [19][20][21][22].
7. The shaft for the wheels 5 mm
8. A knuckle bracket was used with the shaft to mount the front wheels
9. 4-40 Washers, Hex Nuts, and Screws were used to mount all components

10. A 7/8” Length of sides bracket was used to mount the servo mount board. This used No.8 size screws.
11. For additional electronics we used an Arduino Mega, Arduino Uno, Dual MC 33926 Motor Driver from Pololu, Adafruit 9-DOF Absolute Orientation IMU Fusion Breakout – BNO055 Gyroscope, HC-SR04 ultrasonic sensors, and 3D robotics 11.1 V battery for the motors. [23][24][25].
12. Components are wired based on datasheets, by using a breadboard, and by following pinouts on arduino codes mentioned in Appendix A.
13. Loose components were held in place with Velcro

References:

[1] Cbsnews.com, 'Google driverless cars hit the road, almost hit Delphi driverless car', 2015

[2]R. Verma, "Development of a Scaled Vehicle with Longitudinal Dynamics of an HMMWV for an ITS Testbed", *IEEE/ASME Transactions on Mechatronics*, vol. 13, no. 1, 2016.

[3]"Multipurpose 6061 Aluminum", *McMaster-Carr*, 2016. [Online]. Available: http://www.mcmaster.com/#standard-aluminum-sheets/=11qkarp. [Accessed: 28- Mar- 2016].

[4]"Easy-to-Machine Impact-Resistant ABS", *McMaster-Carr*, 2016. [Online]. Available: http://www.mcmaster.com/#standard-plastic-sheets/=11qkbku. [Accessed: 28- Mar- 2016].

[5]*Vehicle Dynamics Modeling*, 1st ed. 2016, p. 10.

[6]*Preventing Slips, Trips, and Falls in Wholesale and Retail Trade Establishments*, 1st ed. 2016.

[7]"Pololu - 19:1 Metal Gearmotor 37Dx52L mm with 64 CPR Encoder (No End Cap)", *Pololu.com*, 2016. [Online]. Available: https://www.pololu.com/product/1442. [Accessed: 06- Apr- 2016].

[8]*LV - MaxSonar - EZ Series*, 1st ed. 2016

[9]*HC-SR04 User Guide*, 1st ed. 2016.

[10]M. Obrist, "Flexible bat echolocation: the influence of individual, habitat and conspecifics on sonar signal design", *Behavioral Ecology and Sociobiology*, vol. 36, no. 3, p. 210, 2016.

[11]Z. Zhu, "Model Predictive Control for Lane Change of Autonomous Vehicle", 2016.

[12]"Arduino - ArduinoBoardMega2560", *Arduino.cc*, 2016. [Online]. Available: https://www.arduino.cc/en/Main/ArduinoBoardMega2560. [Accessed: 28- Mar- 2016].

[13]U. Ozguner, T. Acarman and K. Redmill, *Autonomous Ground Vehicles*. 2016, pp. 167-172.

[14]"GM to Acquire Cruise Automation to Accelerate Autonomous Vehicle Development", *Gm.com*, 2016. [Online]. Available: https://www.gm.com/mol/m-2016-mar-0311-cruise.html. [Accessed: 28- Mar- 2016].

[15]"Arduino Playground - NewPing Library", *Playground.arduino.cc*, 2016. [Online]. Available: http://playground.arduino.cc/Code/NewPing. [Accessed: 06- Apr- 2016].

[16]"Arduino Playground - MaxSonar", *Playground.arduino.cc*, 2016. [Online]. Available: http://playground.arduino.cc/Main/MaxSonar. [Accessed: 06- Apr- 2016].

[17]"Pololu Stamped Aluminum L-Bracket Pair for 37D mm Metal Gearmotors", *Pololu.com*, 2016. [Online]. Available: https://www.pololu.com/product/1084. [Accessed: 06- Apr- 2016].

[18]"Pololu - Power HD High-Torque Servo 1501MG", *Pololu.com*, 2016. [Online]. Available: https://www.pololu.com/product/1057. [Accessed: 06- Apr- 2016].

[19]"17mm Hex Wheel Adaptors", *Servocity.com*, 2016. [Online]. Available: https://www.servocity.com/html/17mm_hex_wheel_adaptors.html#.VgvnKxNViko. [Accessed: 06- Apr- 2016].

[20]"0.770" Clamping Hubs", *Servocity.com*, 2016. [Online]. Available: https://www.servocity.com/html/0_770__clamping_hubs.html#.VgvozhNViko. [Accessed: 06- Apr- 2016].

[21]"0.770" Set Screw Hubs", *Servocity.com*, 2016. [Online]. Available: https://www.servocity.com/html/0_770__set_screw_hubs.html#.VgvpAhNViko. [Accessed: 06- Apr- 2016].

[22]"Amazon.com: Skyq 4pcs RC 1/8 Scale Off Road Car Buggy RC Tires Tyre and Wheels for Redcat HSP HPI Black: Toys & Games", *Amazon.com*, 2016. [Online]. Available: http://www.amazon.com/Scale-Buggy-Tires-Wheels-Redcat/dp/B013U4SGGW/ref=sr_1_13?s=toys-and-games&ie=UTF8&qid=1443044011&sr=1-13&keywords=rc+wheels+1+8+scale. [Accessed: 06- Apr- 2016].

[23]"Pololu - Dual MC33926 Motor Driver Carrier", *Pololu.com*, 2016. [Online]. Available: https://www.pololu.com/product/1213. [Accessed: 06- Apr- 2016].

[24]A. Industries, "Adafruit 9-DOF Absolute Orientation IMU Fusion Breakout - BNO055 ID: 2472 - $34.95 : Adafruit Industries, Unique & fun DIY electronics and kits", *Adafruit.com*, 2016. [Online]. Available: https://www.adafruit.com/products/2472. [Accessed: 06- Apr- 2016].

[25]"IRIS+ Quadcopter 11.1V 5100mAh LiPo Battery", *Robotshop.com*, 2016. [Online]. Available: http://www.robotshop.com/en/iris-quadcopter-111v-5100mah-lipo-battery.html. [Accessed: 06- Apr- 2016].